
FROM DDD TO DELIVERY
Daniel Hinojosa


DOMAIN DRIVEN DESIGN

WHAT IS IT?

- Domain Driven Design (DDD) is a set of tools that assist you in designing and implementing software that delivers high value, both *strategically* and *tactically*
- DDD focus your attention on what your company or your team should be focused on
- DDD has a strategic value in about mapping business domain concepts into software artifacts.
- DDD is about organizing code artifacts in alignment with business problems, using the same common, ubiquitous language
- DDD is about taking this business language, the ubiquitous language and creating models that represent the language and used to solve problems

WHAT IS IT NOT?

- DDD isn’t a methodology, it’s more about the software’s architectural design, providing a structure of practices to take design decisions that help in software projects that have complicated domains.

WAIT, ISN'T THIS WATERFALL?

- Domain-Driven Design encourages incremental development, not waterfall one.
- DDD is about the understanding of the complex domain, and it simply cannot be fully discovered at one go.
- DDD is incremental, learned, and adjusted over time

ADVANTAGES, DISADVANTAGES

- Advantages
  - Incremental & Flexible
  - Organized: In code and teams
  - Focuses on the Domain
- Disadvantages
  - Learning Curve
  - Time and Effort

TWO PARTS OF DDD

Strategic design

Answering the questions of “what?” and “why?”—what software we are building and why we are building it.

Tactical design

About the “how”—how each component is implemented.

ARE THERE TIMES WHEN I CANNOT USE DDD?

- This is a difficult question, and in the end it will be up to you and your team
- Some things to consider:
  - Very simple projects where the domain is one to maybe ten different models. It might be overkill to use DDD (Opinion)
  - Regardless, thinking about domains and models is essential even if you and your team are not going through the entire DDD process
  - If you are in a large company, with many business roles, then DDD will give benefit and clarity
  - In the end, it depends, but you will find many aspects of DDD helpful in the large

FREE REFERENCE

Download a free DDD Reference from the DomainLanguage.com website

DOMAIN DRIVEN DESIGN CREW

Visit the Domain Driven Design Crew's website for starter kits and templates

STRATEGIC


SUBDOMAINS [STRATEGIC]

UNDERSTANDING THE PROBLEM

- "Do we need to know this material? We are writing software, not running businesses."
- The answer to their question is a resounding "yes."
- To design and build an effective solution, you have to understand the problem.
- The problem, in our context, is the software system we have to build.
- To understand the problem, you have to understand the context within which it exists:
  - The organization's business strategy
  - What value it seeks to gain by building the software.

BUSINESS DOMAIN

- Defines a company's main area of activity, it's *primary business*.
- Generally speaking, it's the service the company provides to its clients.
- Examples include:
  - FedEx provides courier delivery.
  - Starbucks is best known for its coffee.
  - Walmart is one of the most widely recognized retail establishments

ANCILLARY DOMAINS

- A company can operate in multiple business domains.
- For example:
  - Amazon provides both retail and cloud computing services.
  - Uber is a rideshare company that also provides food delivery and bicycle-sharing services.
- **It's important to note that companies may change their business domains often!**
- Some examples:
  - Nokia - started off as a paper mill
  - Nintendo - stared off as a playing card service
  - Sony - Radio repair shop
  - McDonalds - Sold hot dogs

SUBDOMAINS

- For a business domain's goals and targets, a company has to operate in *multiple subdomains*
- A *subdomain* is a fine-grained area of business activity
- All of a company's subdomains form its business domain: the service it provides to its customers
- The subdomains have to interact with each other to achieve the company's goals in its business domain

REAL WORLD SUBDOMAINS

- Starbucks may be most recognized for its coffee
- The successful coffeehouse chain requires more than just knowing how to make great coffee.
- It also includes:
  - How to buy or rent real estate at effective locations
  - Hire personnel
  - Manage finances
  - Other activities
- All subdomains are necessary for the complete business

TYPES OF SUBDOMAINS

- Core
- Generic
- Supporting

CORE SUBDOMAIN

- A core subdomain is what a company does differently from its competitors
  - "Inventing new products or services"
  - "Reducing costs by optimizing existing processes"
  - "Ship products to hard-to-reach locations"

CORE SUBDOMAIN COMPLEXITY

- "A simple core subdomain is short-lived competitive advantage"
- Core subdomains are:
  - Naturally complex
  - Hard for competitors to copy or imitate
- Although core subdomains are complex, they don’t necessarily need to be technical

GENERIC SUBDOMAINS

- Generic subdomains are business activities that all companies are performing in the same way.
- Like core subdomains, generic subdomains are generally complex and hard to implement.
- However, generic subdomains do not provide any competitive edge for the company.
- There is no need for innovation or optimization here: battle-tested implementations are widely available, and all companies use them.

GENERIC SUBDOMAIN DISTILLED

- For example, most systems need to authenticate and authorize their users.
- Instead of inventing a proprietary authentication mechanism, it makes more sense to use an existing solution.
- Such a solution is likely to be more reliable and secure since it has already been tested by many other companies that have the same needs.

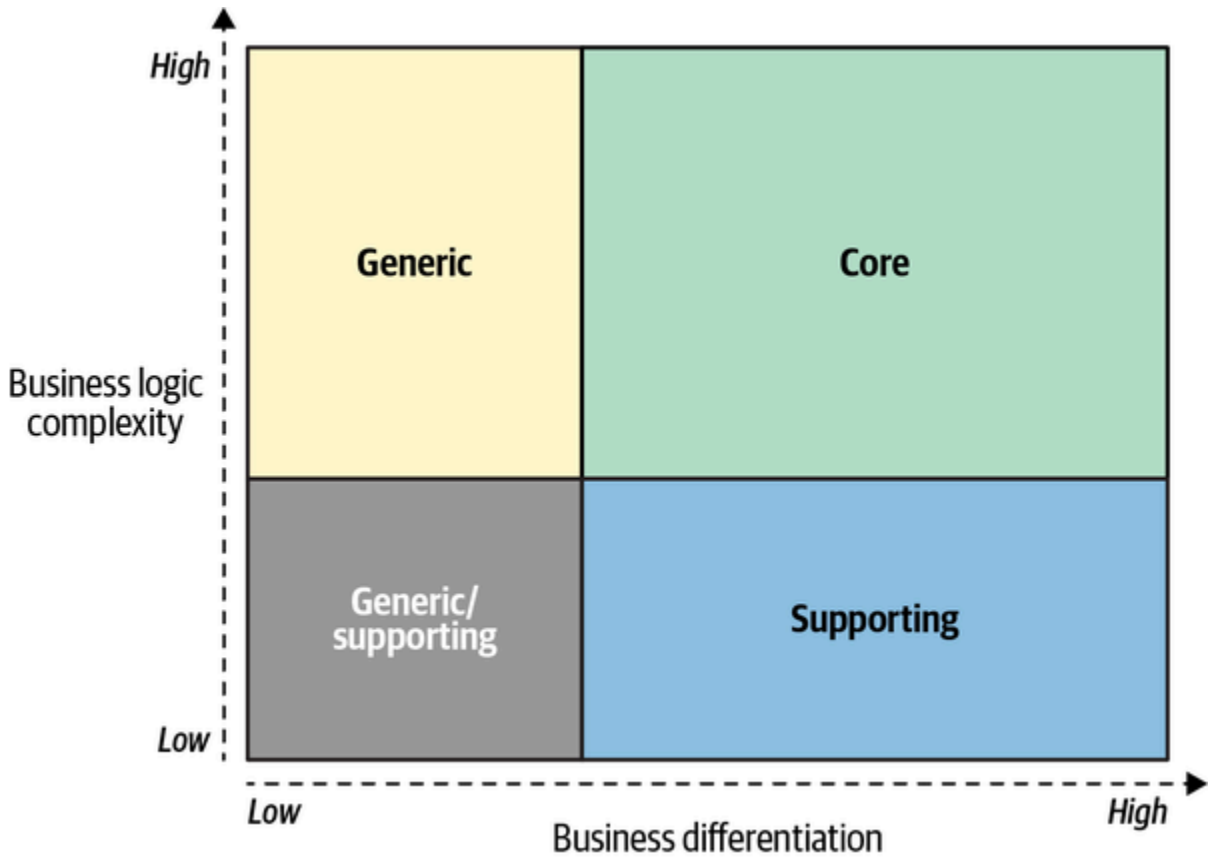
SUPPORTING SUBDOMAIN

- Supporting subdomains support the company's business.
- However, contrary to core subdomains, supporting subdomains do not provide any competitive advantage.

SUPPORTING SUBDOMAIN DISTILLED

- Supporting subdomains are simple.
- Their business logic resembles mostly data entry screens and ETL (extract, transform, load) operations; that is, the so-called CRUD (create, read, update, and delete) interfaces.
- These activity areas do not provide any competitive advantage for the company, and therefore do not require high entry barriers.

SUBDOMAIN DIFFERENTIATION



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

VOLATILITY IN SUBDOMAINS

Core Subdomains

Change is often. If it can be solved in the first attempt it is not competitive. *Evolution is constant*.

Supporting Subdomains

Do not change often. Does not provide any advantage. *Evolution is minuscule*.

Generic Subdomains

The changes can come in the form of security patches, bug fixes, or entirely new solutions to the generic problems.

IMPLEMENTATION IN SUBDOMAINS

Core Subdomains

Core subdomains have to be implemented in-house. They cannot be bought or adopted; that would undermine the notion of competitive advantage. They cannot also be outsourced. They cannot be done by cutting corners. The most skilled talent works on the core subdomain. Should be easy to evolve.

Supporting Subdomains

Rarely any supporting solutions are available but are often in-house. Supporting subdomains do not require highly skilled technical aptitude and provide a great opportunity to train up-and-coming talent.

Generic Subdomains

Buy an off-the-shelf product or adopt an open source solution

TABLE COMPARING THE DOMAINS

Subdomain type	Competitive advantage	Complexity	Volatility	Implementation	Problem
Core	Yes	High	High	In-house	Interesting
Generic	No	High	Low	Buy/adopt	Solved
Supporting	No	Low	Low	In-house/outsource	Obvious

IDENTIFYING DOMAIN EXPERTS

- *Domain experts* **are**:
  - Subject-matter experts who know all the intricacies of the business that we are going to model and implement in code.
  - Knowledge authorities in the software's business domain
- *Domain experts* **are not**:
  - The analysts gathering the requirements
  - The engineers designing the system

DOMAIN EXPERTS DISTILLED

- Domain experts are either the people coming up with requirements or the software's end users
- The software is supposed to solve *their problems*

DOMAIN EXPERTS AND SCOPE

- The domain experts' expertise can have different scopes:
  - Some subject-matter experts will have a detailed understanding of how the entire business domain operates
  - Others will specialize in particular subdomains.
- For example, in an online advertising agency, the domain experts would be:
  - Campaign managers
  - Media buyers
  - Analysts
  - Other business stakeholders


UBIQUITOUS LANGUAGE [STRATEGIC]

GRASPING KNOWLEDGE OF THE BUSINESS DOMAIN

- We are perhaps not experts in the domain that we are designing for
- We have to mimic the way that the experts perform their jobs
- We have to understand the problem in order to solve it

COMMUNICATION

- Success depends on collaboration of stakeholders:
  - Product Owners
  - Domain Experts
  - Engineers
  - UI and UX Designers
  - Project Managers
  - etc.

ISSUES THAT ARISE WITH COMMUNICATION

- Do all stakeholders agree on what problem is being solved?
- Do they hold any conflicting assumptions about its functional and nonfunctional requirements?

MEDIATORS IN COMMUNICATION

- Communication is essential for success, but it is rarely observed
- Business people and software engineers have no direct interaction with one another
- Unfortunately communication from domain experts is filtered by "mediators and translators"

WHO ARE THE TRANSLATORS?

- Systems/business analysts
- Product owners
- Project managers

ANALYSIS MODEL

- Taking *domain knowledge* and translating it into engineering friendly *analysis model*
- The analysis model is the description of the system's requirements rather than understanding the business domain
- Mediation can be hazardous, domain knowledge that is essential for solving business problems gets lost on its way to the software engineers
- Information can be distorted

UBIQUITOUS LANGUAGE

- Cornerstone of Domain Driven Design
- A dictionary of the same language
- A language for describing your business domain
- Represents both the business domain and the domain experts' mental models.

UBIQUITOUS LANGUAGE GOALS

- Represent the companies terms only
- No technical jargon (e.g. RestFUL, Patterns, Kubernetes)
- Frame the domain experts' understanding and mental model
- Make it easy to understand and eliminate assumptions

EXAMPLE OF UBIQUITOUS LANGUAGE

- "An advertising campaign can display different creative materials."
- "A campaign can be published only if at least one of its placements is active."
- "Sales commissions are accounted for after transactions are approved."

EXAMPLE OF POOR UNDERSTANDING OF UBIQUITOUS LANGUAGE

- "The advertisement iframe displays an HTML file."
- "A campaign can be published only if it has at least one associated record in the active-placements table."
- "Sales commissions are based on correlated records from the transactions and approved-sales tables."

CONSISTENCY

- Each term of the Ubiquitous Language should have only *one meaning*
- If it doesn't it may fall into one of the following categories:
  - Ambiguous Terms
  - Synonymous Terms

AMBIGUOUS TERMS

- Example: In business domain, the term policy has multiple meanings:
  - A regulatory rule
  - An insurance contract
- Ubiquitous language demands a single meaning for each term
- “policy” should be modeled explicitly using the two terms *regulatory rule* and *insurance contract*

SYNONYMOUS TERMS

- Two terms cannot be used interchangeably in a ubiquitous language
- Example: *user*
- User may be used interchangeably: user, visitor, administrator, account, etc.

WHAT IS A USER?

- One term can denote different concepts
- *visitor*, *account* can refer to system users
- uregistered and registered users represent different roles and different behaviors
- Visitors are for analysis; Accounts are use the systems

UBIQITUOUS LANGUAGE AND ARTIFACTS

Ubiquitous Language should be used in

- Requirements
- Tests
- Documentation
- Source Code

UBIQUITOUS LANGUAGE IN A GLOSSARY

- A *wiki* to be used as a glossary to capture the ubiquitous language
  - It makes it easy to onboard new members with terms
  - It is a shared resource
  - Include nouns: names of entities, processes, roles, etc.
  - Include verbs: processes and behaviour

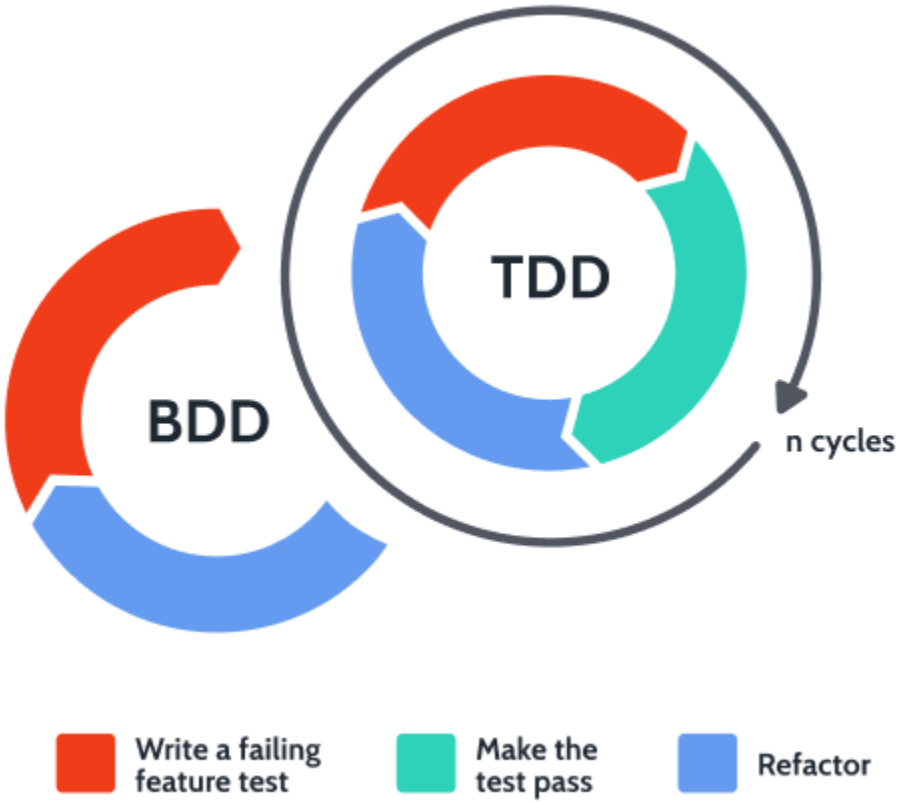
USING GHERKIN AND BDD TO DERIVE TESTING

```Gherkin
Given the account has a balance of $23.95
When a message of "balance" comes in from a 404-333-2222
Then a text is sent back: "Your balance is $23.95"

Given the account has a balance of $23.95
When any message comes in from an unknown number 233-123-1121
Then log message and phone number, and do nothing

Given the account has a balance of $23.95
When any message other than "balance" comes in from a 404-333-2222
Then send text back: "Didn't understand message, send BALANCE for current balance."
```

BDD AND TDD


TDD
BDD
n cycles
Write a failing feature test
Make the test pass
Refactor


BOUNDED CONTEXTS [STRATEGIC]

CONSISTENT & INCONSISTENT MODELS

- Ubiquitous Language should be consistent since it will drive decisions
- As companies scale, the domain experts' mental models will be inconsistent

INCONSISTENT MODELS

What if after examining domain expert's language, you find that in two context there is overloaded term: *lead*?

Marketing Department

For the marketing people, a lead represents a notification that somebody is interested in one of the products. The event of receiving the prospective customer's contact details is considered a lead

Sales Department

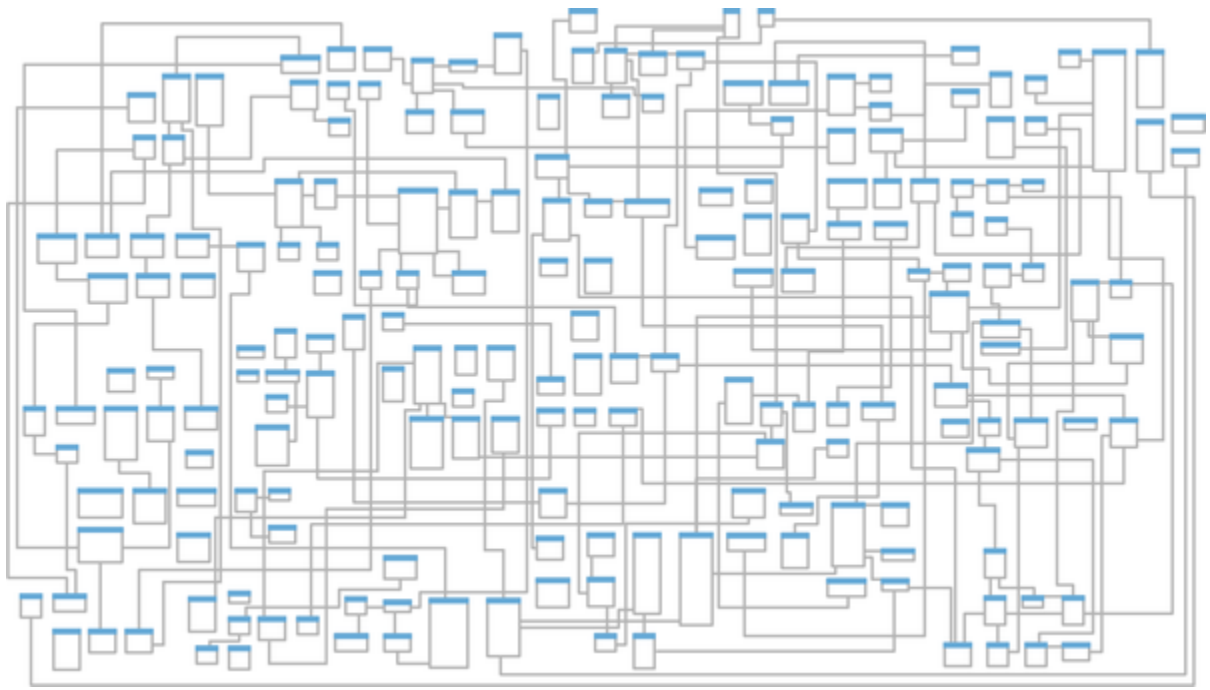
In the context of the sales department, a lead is a much more complex entity. It represents the entire lifecycle of the sales process. It's not a mere event, but a long-running process

DEALING WITH AMBIGUITY

- Ubiquitous Language needs to be consistent - each term should have one meaning
- Ubiquitous Language needs to map to domain experts' mental models
- Having two *lead*s isn’t a problem *in communication*, but it is in software

THE TRADITIONAL APPROACH

- The Traditional Approach for such modeling a single model
- Typically defined by an enormous entity relationship (ERD)
- "Suitable for everything but eventually are effective for nothing"



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

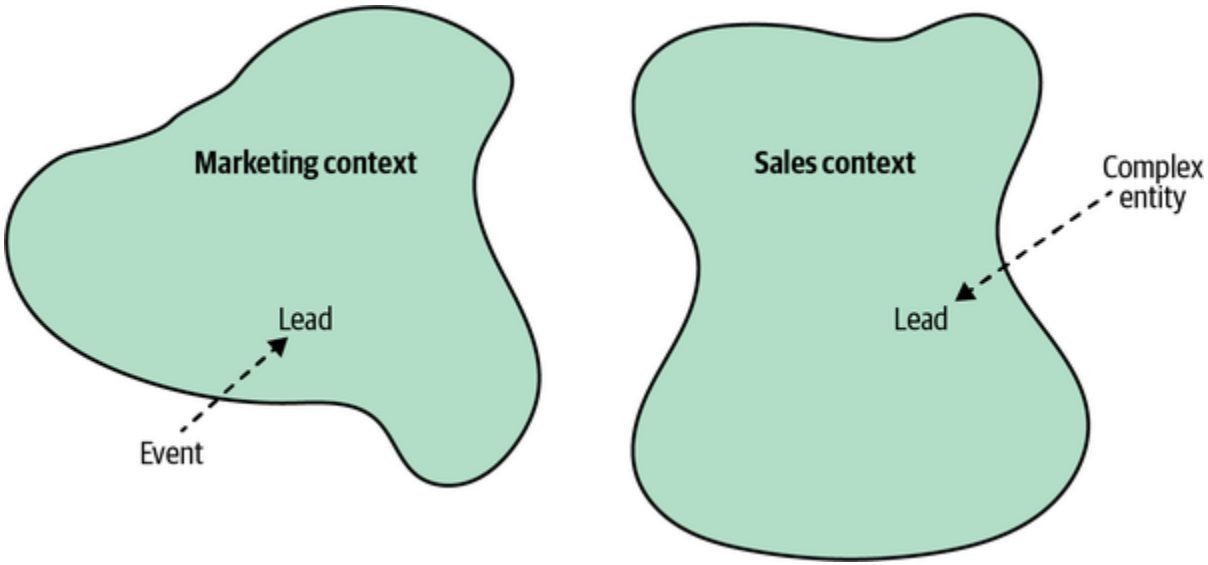
WHAT ABOUT PREFIXING?

- Can we do “marketing lead” and “sales lead"?
- Allows for two different models of code
- This has three main disadvantages:
  - Creates cognitive load. When should each be used? The closer they are the likely a mistake
  - The implementation will not be aligned with the ubiquitous languages
  - Does anyone use the prefixed terms in everyday language?

INTRODUCING THE BOUNDED CONTEXT

- The solution:
  - *Divide the Ubiquitous Language into multiple smaller languages*
  - Assign each team to the explicit context in which it can be applied: its *bounded context*

BOUNDED CONTEXT BY EXAMPLE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

MODEL BOUNDARIES

- A model is a construct that helps us makes sense of a complex system
- It is supposed to solve an inherent part of a model - it’s purpose
- A model cannot exist without a boundary

PURPOSE FOR BOUNDARIES

- Just like types of maps - aerial, nautical, terrain, subway, and so on
- One bounded context can be completely irrelevant to the scope of another
- They allow distinct models according to different problem domains
- Bounded contexts are the consistency boundaries of ubiquitous language
- A Ubiquitous Language is *not universal*

UBIQUITOUS WITHIN A CONTEXT

- A ubiquitous language is ubiquitous only in the boundaries of its bounded context
- The language is focused on describing only the model that is encompassed by the bounded context
- A ubiquitous language cannot be defined or used without an explicit context of its applicability

WE CAN NARROW THE SCOPE OF A BOUNDED CONTEXT

- You can decompose bounded contexts into smaller bounded contexts, based on language and relevancy
- Defining bounded contexts is a strategic design decision
- Boundaries can be:
  - **Wide** - Following the business domain's inherent contexts
  - **Narrow** - Further dividing the business domain into smaller problem domains

"Models shouldn't necessarily be big or small. Models need to be useful."

SO WHAT IS IT? SMALL OR LARGE BOUNDED CONTEXT?

- The wider the boundary of the ubiquitous language is, the harder it is to keep it consistent.
- It may be beneficial to divide a large ubiquitous language into smaller, more manageable problem domains
- But, striving for small bounded contexts can backfire too.
- The smaller they are, the more integration overhead the design induces
- The decision, is based on your problem domain

MORE REASONS FOR FINE-GRAINED BOUNDED CONTEXTS

- Creating new software engineering teams or addressing some of the system's nonfunctional requirements; for example, when you need to separate the development lifecycles of some of the components originally residing in a single bounded context
- Ability to scale it independently from the rest of the bounded context's functionalities

MORE REASONS FOR COARSE-GRAINED BOUNDED CONTEXTS

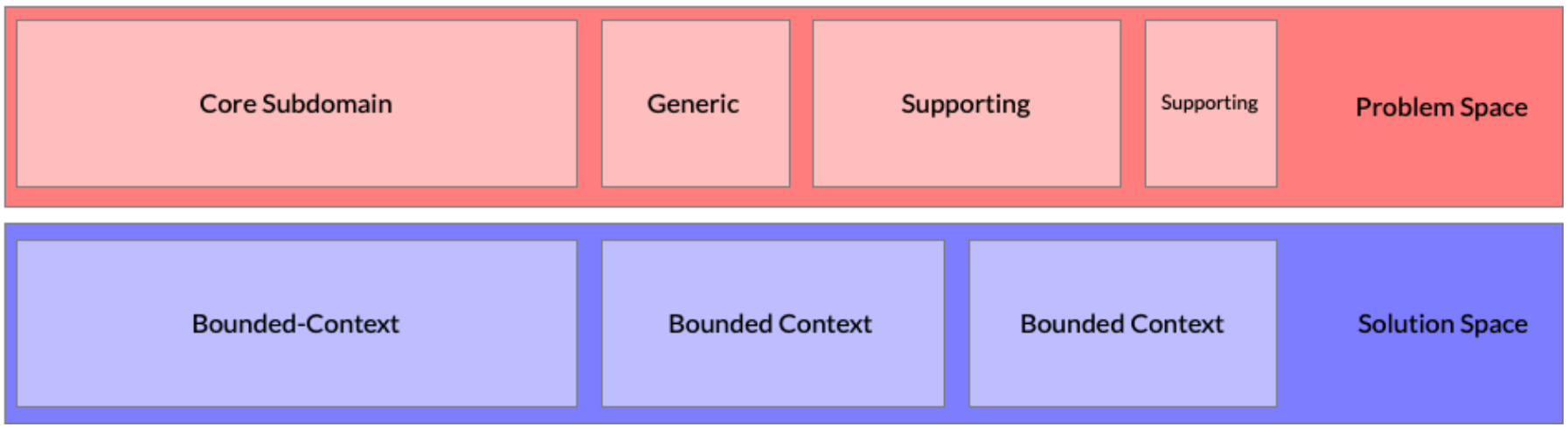
- Division will hinder the ability to evolve each context independently.
- Instead, the same business requirements and changes will simultaneously affect the bounded contexts and require simultaneous deployment of the changes.
- Identify sets of coherent use cases that operate on the same data and avoid decomposing them into multiple bounded contexts.

BOUNDED CONTEXTS VS SUBDOMAINS

- Subdomains
  - Recall identification of different subdomains: core, supporting, generic
  - Subdomains are interrelated use-cases and defined by the business domain and system requirements
- Bounded Contexts
  - Designed - A strategic designed decision
  - We decide how to divide the business domain into smaller, manageable problem domains

BOUNDED CONTEXTS VS SUBDOMAINS

- Subdomains are part of the problem space - **"What is in place and what are we solving"**
- Bounded Contexts are part of the solution space - **"How do we solve the problems"**



BOUNDARIES

The bounded context pattern is the domain-driven design tool for defining boundaries

- Physical
- Ownership

PHYSICAL BOUNDARIES

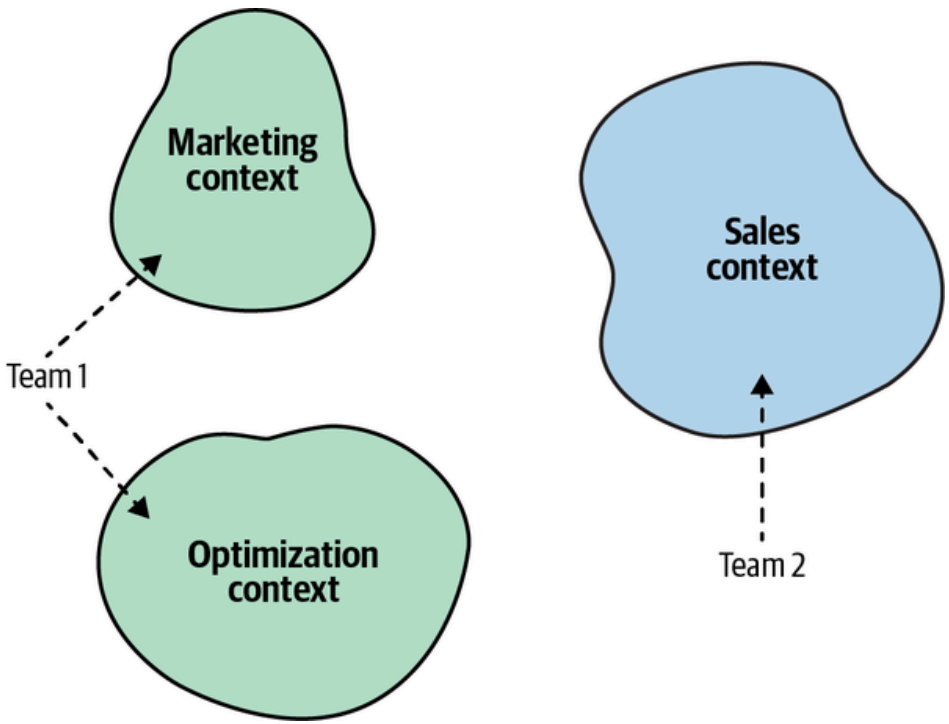
- Each bounded context should be implemented as an individual service/project
  - It is implemented, evolved, and versioned independently of other bounded contexts
- Clear physical boundaries between bounded contexts allow us to implement each bounded context with the technology stack that best fits its needs
- A bounded context can contain multiple subdomains. In such a case, the bounded context is a *physical boundary*, each of its subdomains is a *logical boundary*
- Logical boundaries bear different names in different programming languages: namespaces, modules, or packages.

OWNERSHIP BOUNDARIES

- *A bounded context should be implemented, evolved, and maintained by one team only*
- No two teams can work on the same bounded context
- Segregation eliminates implicit assumptions that teams might make about one another's models

UNIDIRECTIONAL RELATIONSHIP OF BOUNDED CONTEXTS

- A bounded context should be owned by only one team.
- However, a single team can own multiple bounded contexts



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

SEMANTIC DOMAINS

- How you and your team perceive domains is for you to decide.
- Domains are semantic
- e.g. a tomato
  - In botany - the tomato is a fruit
  - In culinary - the tomato is a vegetable
  - In taxation - In 1883 the United States established a 10% tax on imported vegetables, but not fruits, for taxation purposes, *the tomato is a vegetable*
  - In theater - the tomato is a feedback mechanism


INTEGRATING BOUNDED CONTEXTS [STRATEGIC]

CONTRACTS

- A system cannot be built out of independent components, the components have to interact with one another to achieve the system's overarching goals
- The same goes for implementations in bounded contexts.
- Although they can evolve independently, they have to integrate with one another
- These are called *contracts*

CONTRACT BETWEEN CONTRACTS

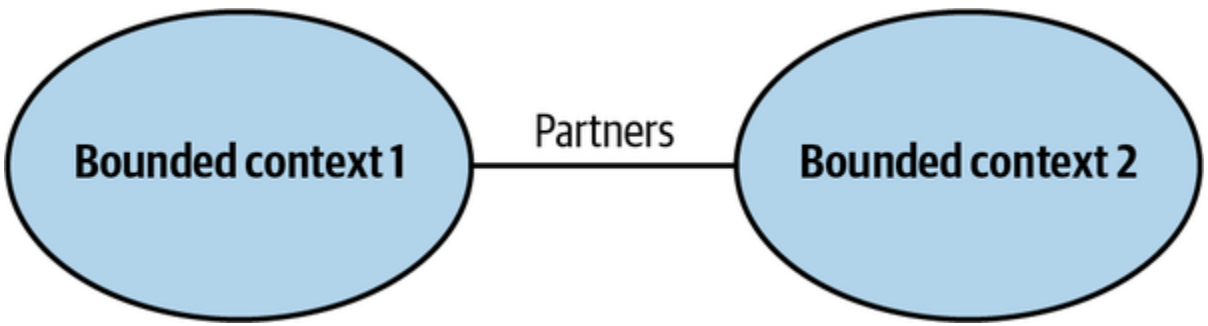
- By definition, two bounded contexts are using different ubiquitous languages.
- Which language will be used for integration purposes?
- These integration concerns should be evaluated and addressed

COOPERATION

- Cooperation patterns relate to bounded contexts implemented by teams with well-established communication.
- They are either:
  - The same team
  - Two teams very tightly coordinated

PARTNERSHIP

- Integration between bounded contexts is coordinated in an ad hoc manner
- One team can notify a second team about a change in the API, and the second team will cooperate and adapt—no drama or conflicts



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

PARTNERSHIP COORDINATION

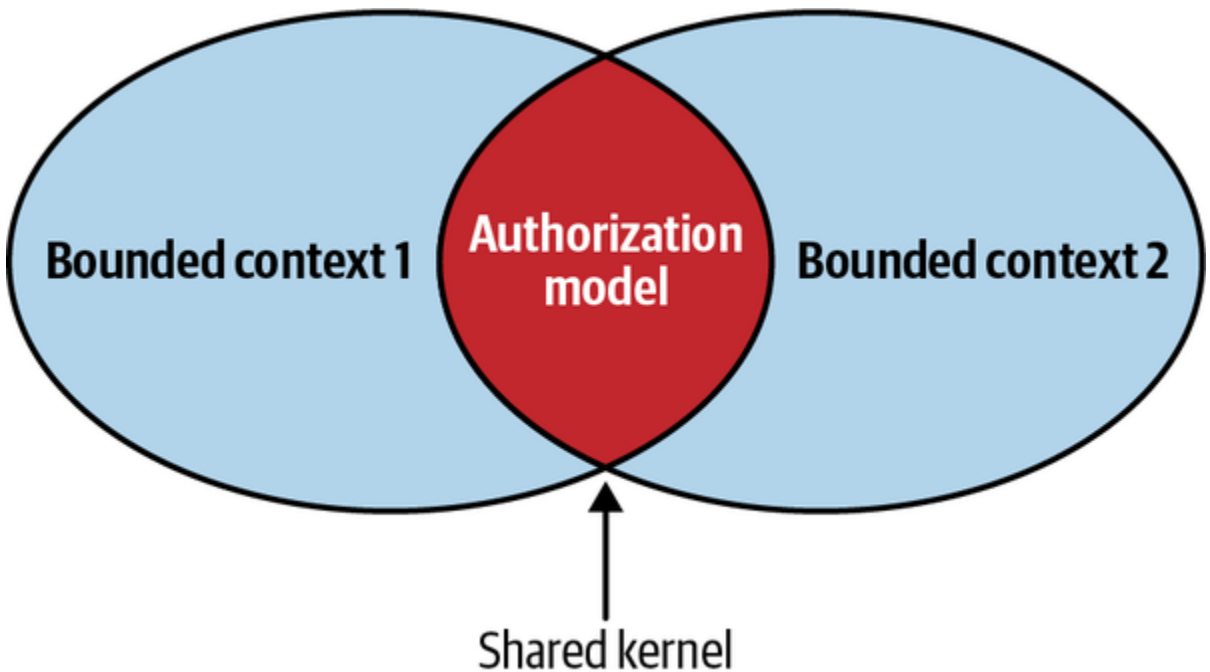
- Coordination is two-way
- No one team dictates the language that is used for defining the contracts
- Teams work out the differences and choose the most appropriate solution
- Both cooperate, and neither is interest in blocking another

PARTNERSHIP REQUIREMENTS

- Commitment
- Constant Communication
- Continuous Integration to minimize the integration feedback loop
- Both teams should highly likely be geographically in the same location

SHARED KERNEL/MODEL

- Same model of a subdomain, or a part of it, implemented in multiple bounded contexts
- Designed according to the needs of all of the bounded contexts
- Consistent across all of the bounded contexts that are using it



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

SHARED KERNEL/MODEL

- Example: Custom Authentication/Authorization module used by separate bounded contexts
- Each bounded context can modify the authorization model, and the changes each bounded context applies have to affect all the other bounded contexts using the model
- To minimize the cascading effects of changes, the overlapping model should expose only that part of the model *that has to be* implemented by both bounded contexts
- Consist only of integration contracts and data structures that are intended to be passed across the bounded contexts' boundaries

IMPLEMENTATION OF SHARED KERNEL

- In a mono-repo: these can be the same source files referenced by multiple bounded contexts
- In a dedicated-repo: shared kernel can be extracted into a dedicated project and referenced in the bounded contexts as a linked library

SHARED KERNEL AND CONTINUOUS INTEGRATION

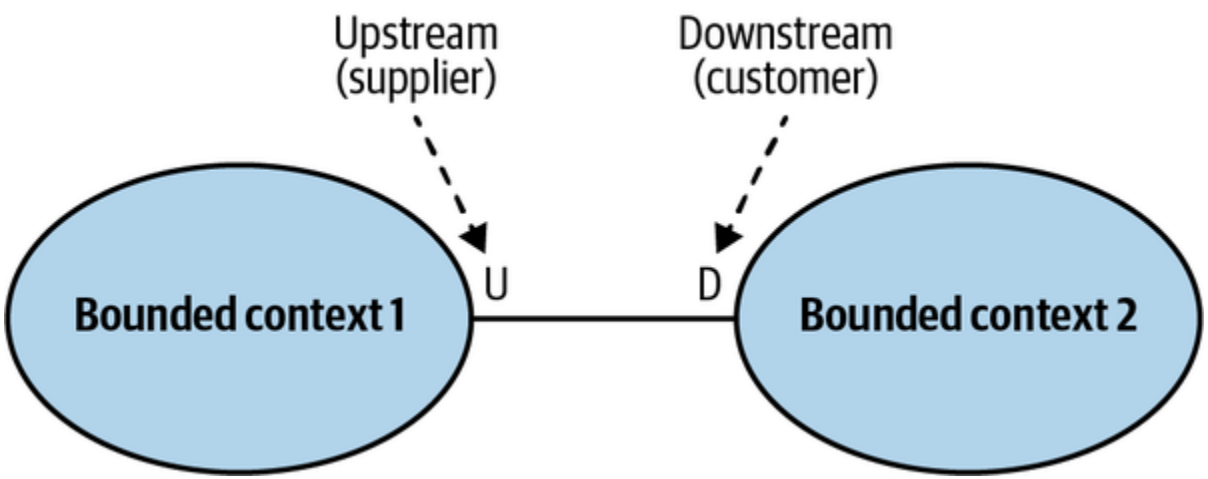
- Continuous Integration of changes is **required** because the shared kernel belongs to multiple bounded contexts
- Not doing so will cause stale implementations leading to data corruption or runtime issues

WHEN TO USE SHARED KERNEL?

- It should be applied only when the *cost of duplication is higher than the cost of coordination*
- In other words, only when integrating changes applied to the shared model by both bounded contexts will require more effort than coordinating the changes in the shared codebase
- Using a shared kernel violates the independence rule of bounded context, and therefore, should be justified
- Shared Kernel can be used as a temporary measure to migrate one legacy application into multiple bounded-contexts
- Shared Kernel can be used as a temporary measure to integrate multiple bounded contexts owned by the same team
- If a lot of model code ends up in the shared kernel, it may be a sign that the contexts should, in fact, be merged into one big context

CUSTOMER-SUPPLIER

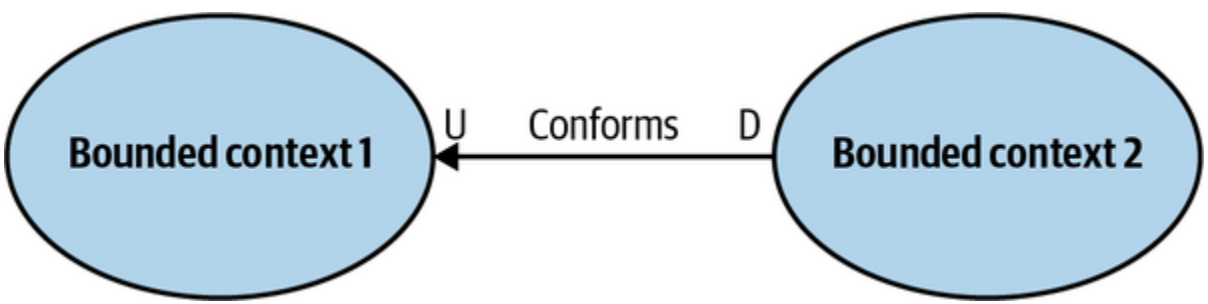
- The service provider is “upstream” and the customer or consumer is “downstream.”
- Both teams (upstream and downstream) can succeed independently
- Imbalance of power: either the upstream or the downstream team can dictate the integration contract
- This is where a decision should be made about the relationship



Learning Domain Driven Design - Vlad Khononov, Published by O’Reilly Media, Inc.

CONFORMIST

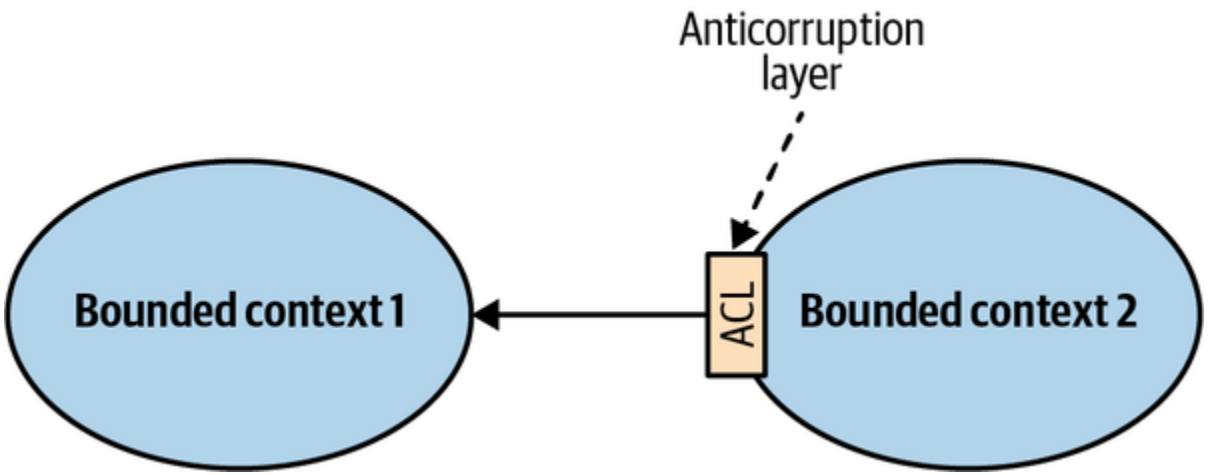
- Balance of power favors the upstream team, no real motivation to support the client
- Upstream provides the integration contract, defined according to its own model—take it or leave it
- Caused by integration with service providers that are external to the organization or simply by organizational politics
- The contract exposed by the upstream team may be an industry-standard, well-established model, or it may just be good enough for the downstream team's needs.



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

ANTI-CORRUPTION LAYER

- The balance of power in this relationship is still skewed toward the upstream service.
- In this case, the downstream bounded context is not willing to conform.
- Instead, it translates the upstream bounded context's model into a model tailored to its own needs via an anticorruption layer



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

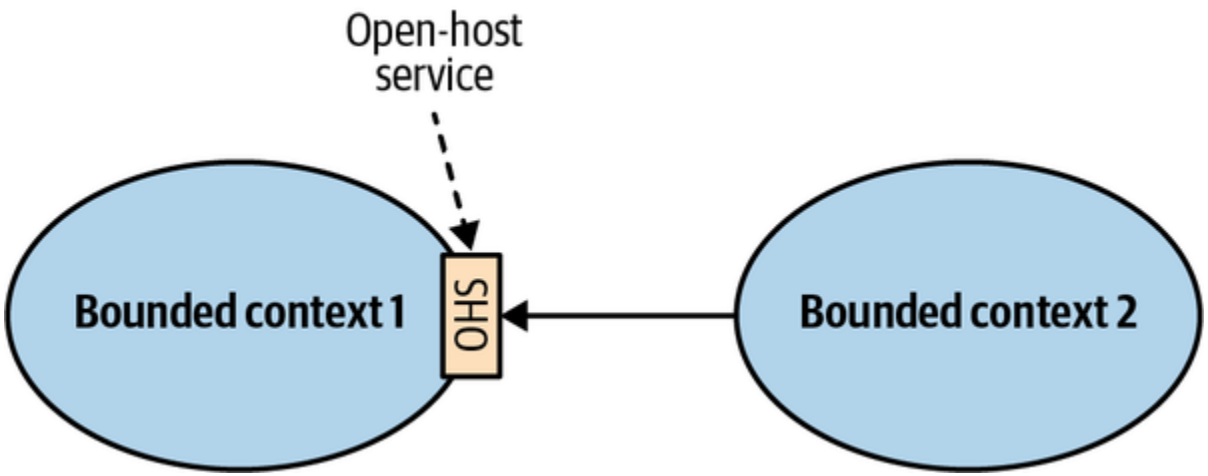
REASONS FOR ANTI-CORRUPTION LAYER

- A core subdomain's model requires extra attention, and adhering to the supplier's model might impede the modeling of the problem domain.
- When the upstream model is inefficient or inconvenient for the consumer's needs
- If a bounded context conforms to a mess, it risks becoming a mess itself (legacy systems)
- When the supplier's contract changes often, the consumer wants to protect its model from frequent changes

OPEN HOST SERVICE

- Power is toward the consumers
- Supplier is interested in protecting its consumers and providing the best service possible.
- Upstream supplier decouples the implementation model from the public interface.
- Allows the supplier to evolve its implementation and public models at different rates
- The supplier exposes a protocol convenient for the consumers

OPEN HOST SERVICE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

PUBLISHED LANGUAGE

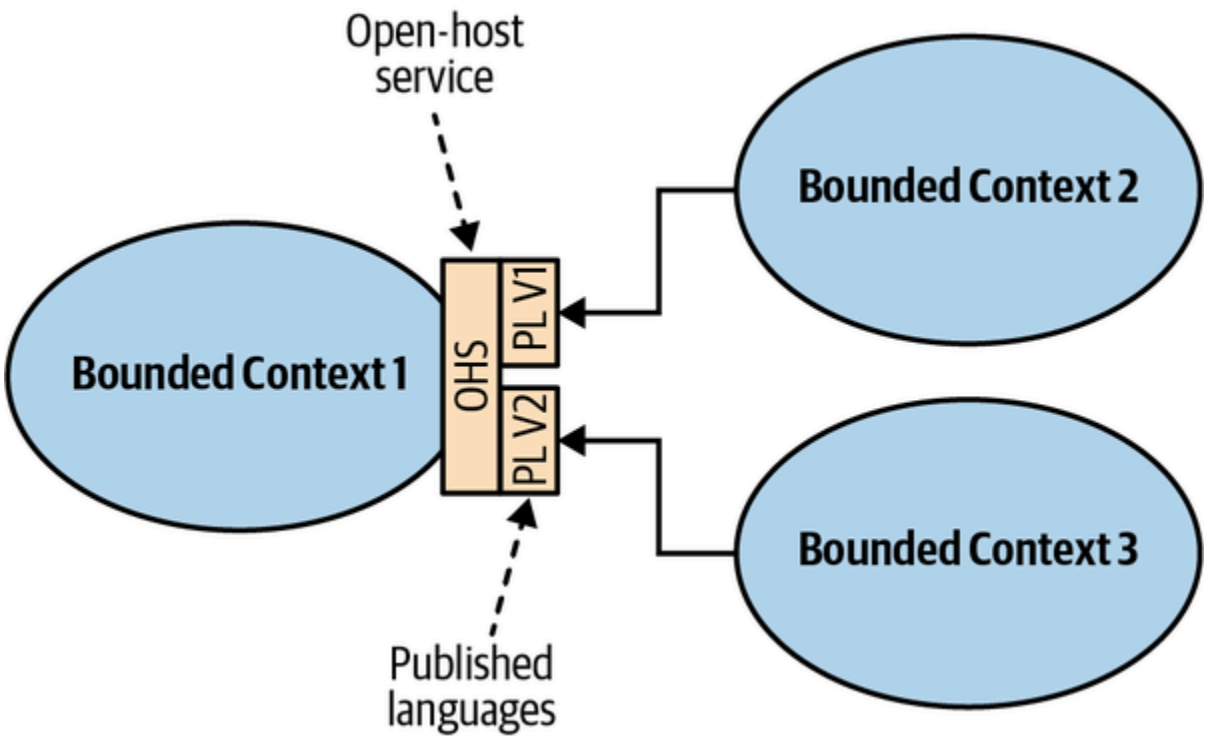
- Publicly used to communicate and integrate between different Bounded Contexts.
- They also have an agreed-meaning across Bounded Contexts.
- Mainly used by engineers working in different Bounded Contexts to agree on integration approach (open-host, client-server, pub-sub, etc.).
- Mainly expressed via technical format (API, JSON, XML, Protocol Buffer, etc.)

OPEN HOST SERVICE

- Upstream bounded context the freedom to evolve its implementation without affecting the downstream contexts
- Possible if the modified implementation model can be translated into the published language the consumers are already using

OPEN HOST SERVICE MULTIPLE PROTOCOLS

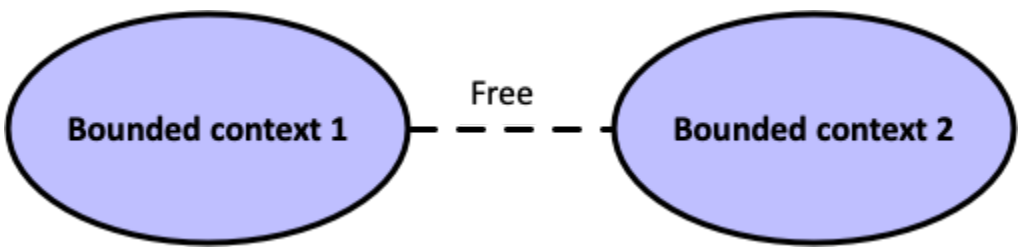
- Integration model's decoupling allows the upstream bounded context to simultaneously expose multiple versions of the published language
- Allowing the consumer to migrate to the new version gradually



SEPARATE WAYS / FREE

Last option or pattern is not to collaborate at all for the following reasons

- Communication Issues
- Generic Subdomains
- Model Differences



COMMUNICATION ISSUES

- Organization Size and Politics hinders collaboration
- Teams may have a hard time collaborating
- It may be more cost-effective to go their separate ways and duplicate functionality in multiple bounded contexts

GENERIC SUBDOMAINS

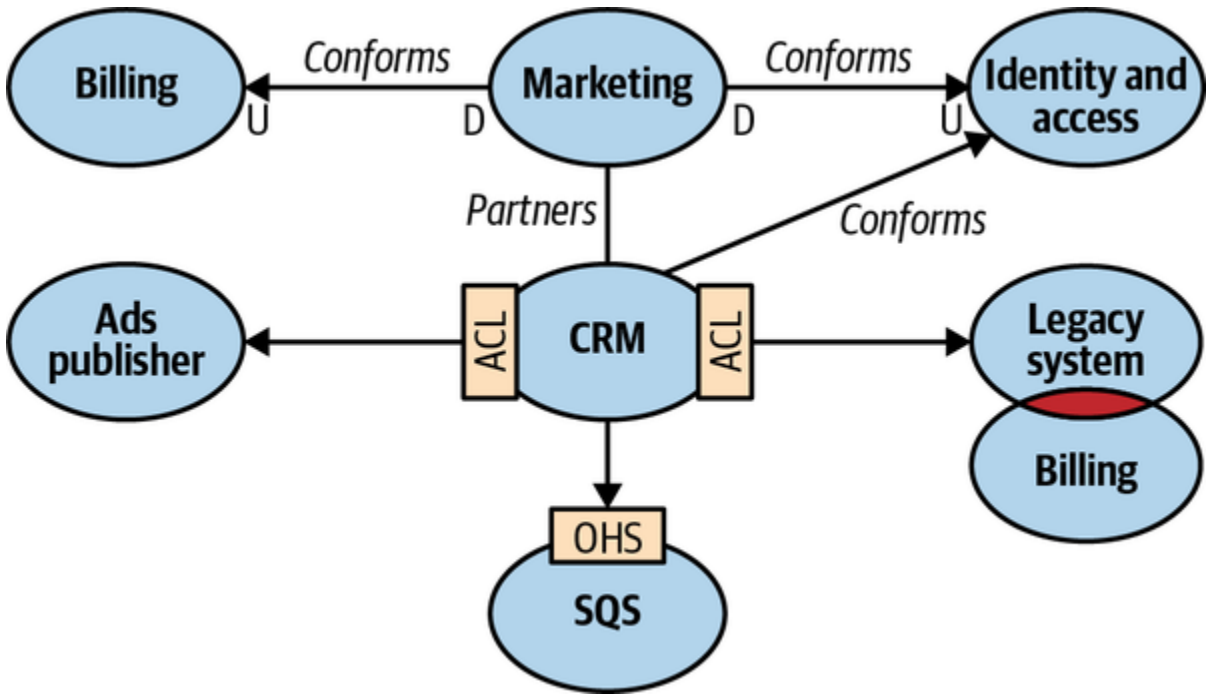
- When the subdomain in question is generic, and if the generic solution is easy to integrate, it may be more cost-effective to integrate it locally in each bounded context.
- An example is a logging framework; it would make little sense for one of the bounded contexts to expose it as a service.
- Duplicating the functionality would be less expensive than collaborating.

MODEL DIFFERENCES

- The models may be so different that a conformist relationship is impossible
- Implementing an anticorruption layer would be more expensive than duplicating the functionality

CREATING A CONTEXT MAP

Context map is a visual representation of the system's bounded contexts and the integrations between them



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

BENEFITS TO A CONTEXT MAP

- Provides an overview of the system's components and the models they implement
- Exhibits the communication patterns among teams and which teams integrate
- Gives insight to organizational issues, forces questions and thought

MAINTENANCE

Context Maps should be

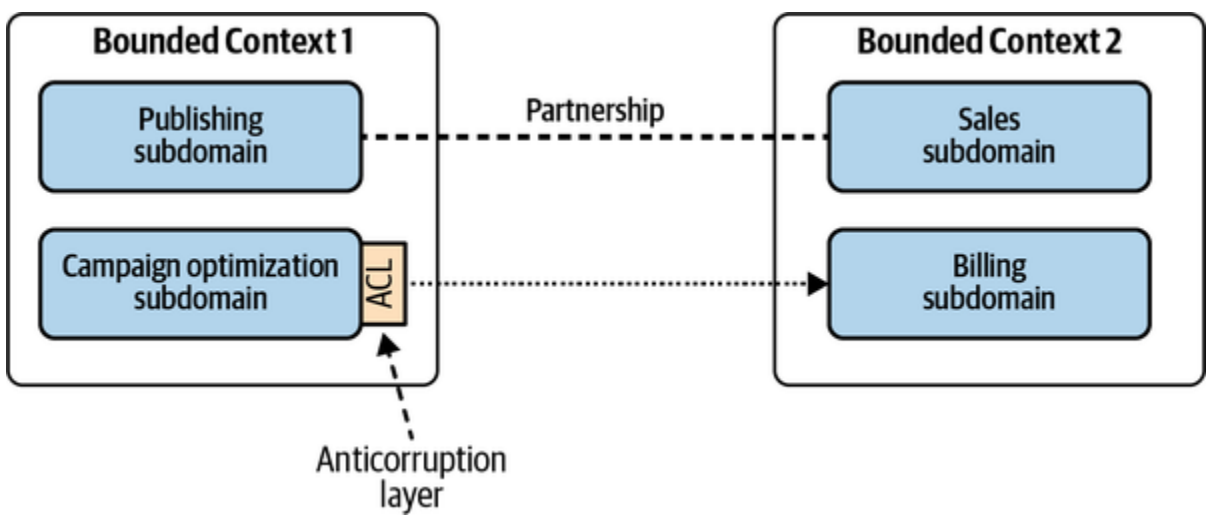
- Introduced from the beginning
- Updated to reflect additions and changes
- A shared effort where each team updates its own integrations

CONTEXT MAPPER

A context map can be managed and maintained as code, using a tool like Context Mapper

LIMITATIONS

- Context Maps can be challenging
- When bounded contexts encompass multiple subdomains, there can be multiple integration patterns at play
- Also, if bounded contexts are limited to a single subdomain, there still can be multiple integration patterns at play
- For example, if the subdomains' modules require different integration strategies.



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

TACTICAL


SIMPLE BUSINESS LOGIC [TACTICAL]

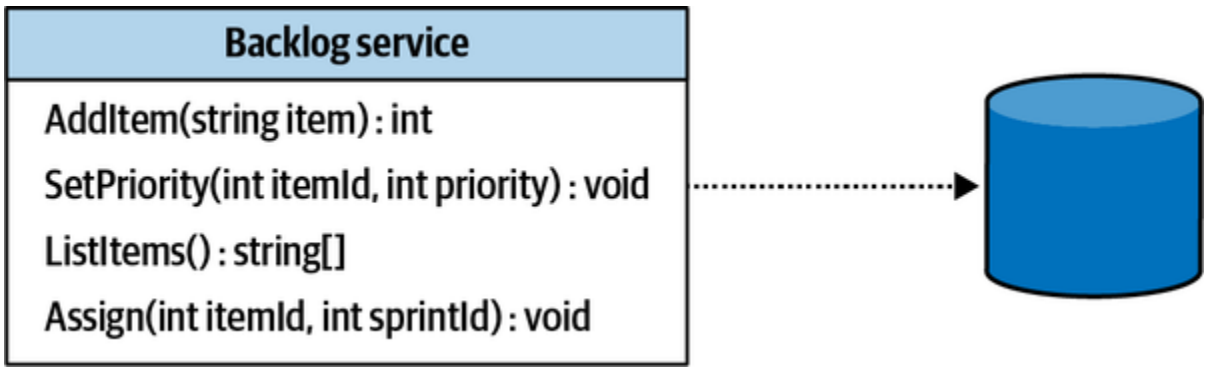
BUSINESS LOGIC

- Business logic is the most important part of software
- You can have the most amazing UI and fast database but if the software is not useful it is a failure
- We will visit two simplistic business logic patterns:
  - Transaction Script
  - Active Record

TRANSACTION SCRIPT

- Organizes business logic by procedures where each procedure handles a single request from the presentation
- A system's public interface can be seen as a collection of business transactions that consumers can execute
- Transactions can retrieve information managed by the system, modify it, or both
- Pattern organizes the system's business logic based on procedures
- Each procedure implements an operation that is executed by the system's consumer via its public interface
- **Public operations are used as encapsulation boundaries**

EXAMPLE OF PUBLIC TRANSACTION SCRIPT INTERFACE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

IMPLEMENTATION OF TRANSACTION SCRIPT

- Each procedure is implemented as a simple, straightforward procedural script
- It can use a thin abstraction layer for integrating with storage mechanisms, but it is also free to access the databases directly
- *Each operation should either succeed or fail but can never result in an invalid state*
- During failure: System should remain consistent—either by rolling back any changes it has made up until the failure or by executing compensating actions

EXAMPLE OF TRANSACTION SCRIPT

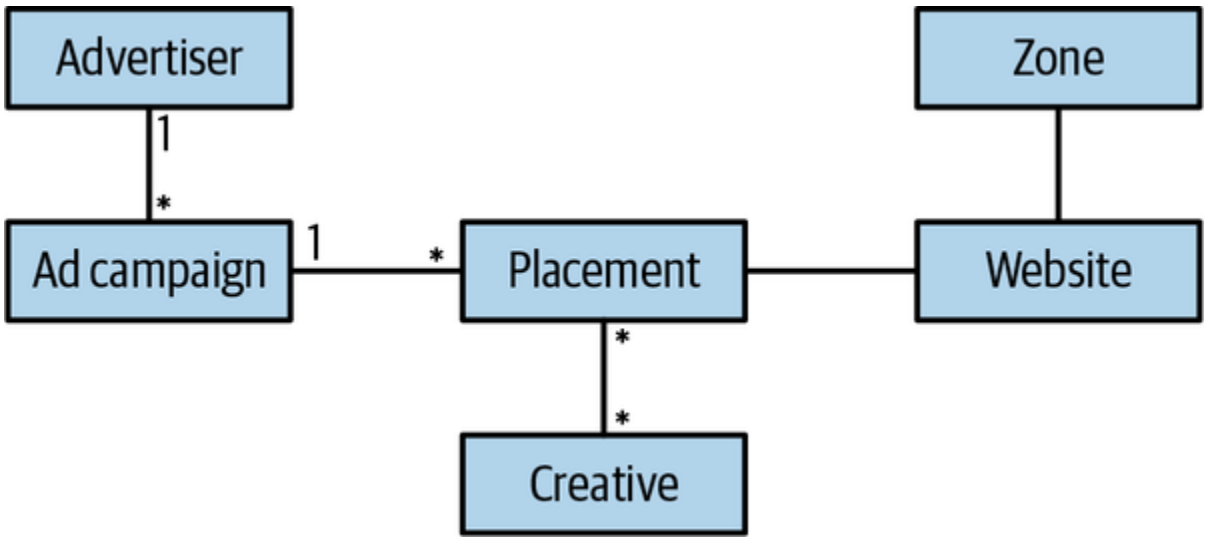
```Java
db.startTransaction();

var job = db.loadNextJob();
var json = loadFile(job.source);
var xml = convertJsonToXml(json);
writeFile(job.Destination, xml.ToString();
db.markJobAsCompleted(job);
db.commit()
```

ACTIVE RECORD

- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data
- Active record supports cases where the business logic is simple
- Business logic may operate on more complex data structures which is different than transaction script
- Active record also supports complicated trees and hierarchies
- Doing something like the following diagram would make it so complicated for something like a transaction script which leads to repetitive code

EXAMPLE OF A COMPLEX ACTIVE RECORD STRUCTURE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

IMPLMENTATION OF ACTIVE RECORD

- Active Record uses dedicated objects, known as active records, to represent complicated data structures
- Active Record objects also implement data access methods for creating, reading, updating, and deleting records—the so-called CRUD operations
- Active record objects are coupled to an object-relational mapping (ORM) or some other data access framework

EXAMPLE OF ACTIVE RECORD

Java

```java
public class CreateUser {
    public void execute(UserDetails userDetails) {
        try {
            db.startTransaction();
            var user = new User();
            user.setName(userDetails.getName());
            user.setEmail(userDetails.getEmail());
            user.save();
            db.commit();
        } catch {
            db.rollback();
            throw new RuntimeException("Unable to persist the Users");
        }
    }
}
```

ADVANTAGES AND DISADVANTAGE TO ACTIVE RECORD

Advantage

- Active record is essentially a transaction script that optimizes access to databases
- Can only support relatively simple business logic, such as CRUD operations, which, at most, validate the user's input

Disadvantage

- It can potentially introduce more harm than good when applied in the wrong context
- It can be considered an anti-pattern, since it mixes two concerns, domain and repository


COMPLEX BUSINESS LOGIC [TACTICAL]

HISTORY

- Transaction script and active record patterns, the domain model pattern was introduced initially in Martin Fowler's book *Patterns of Enterprise Application Architecture*
- Fowler mentioned that "Eric Evans is currently writing a book on building Domain Models" referring to Domain Driven Design that we talk about today
- Eric Evans talked about other patterns: Aggregates, Entities, Value Objects etc. This takes off from Martin Fowler left off.

DOMAIN MODEL

- The domain model pattern is intended to cope with cases of complex business logic.
- Instead of CRUD, we deal with complex state transitions, business rules, and invariants

EXAMPLE COMPLEXITY

1. Customers open support tickets describing issues they are facing.
2. Both the customer and the support agent append messages, and all the correspondence is tracked by the support ticket.
3. Each ticket has a priority: low, medium, high, or urgent.
4. An agent should offer a solution within a set time limit (SLA) that is based on the ticket's priority
5. If the agent doesn't reply within the SLA, the customer can escalate the ticket to the agent's manager.
6. Escalation reduces the agent's response time limit by 33%.

EXAMPLE COMPLEXITY

7. If the agent didn't open an escalated ticket within 50% of the response time limit, it is automatically reassigned to a different agent.
8. Tickets are automatically closed if the customer doesn't reply to the agent's questions within seven days.
9. Escalated tickets cannot be closed automatically or by the agent, only by the customer or the agent's manager.
10. A customer can reopen a closed ticket only if it was closed in the past seven days.

REVIEW THE LIST

- We see that there is a complex list of invariants
- Using Active Records will make this unwieldy and include many duplicates

IMPLEMENTATION OF THE DOMAIN MODEL

- A domain model is an object model of the domain that incorporates both behavior and data
- DDD's tactical patterns — aggregates, value objects, domain events, and domain services— are the building blocks of such an object model.
- All of these patterns share a common theme: they put the business logic first

DON'T ADD MORE COMPLEXITY

- The Domain Objects are already going to model a business process that's already complex, so it shouldn't introduce more complexity
- When defining the domain: **The model should be devoid of any infrastructural or technological concerns, such as implementing calls to databases or other external components of the system. They should not incorporate frameworks or infrastructure**
- They should be *Plain Old Objects*

UBIQUITOUS LANGUAGE

- It is important to **follow the terminology of the bounded context's ubiquitous language**
- Domain Model allows the code to "speak" the ubiquitous language and to follow the domain experts' mental models

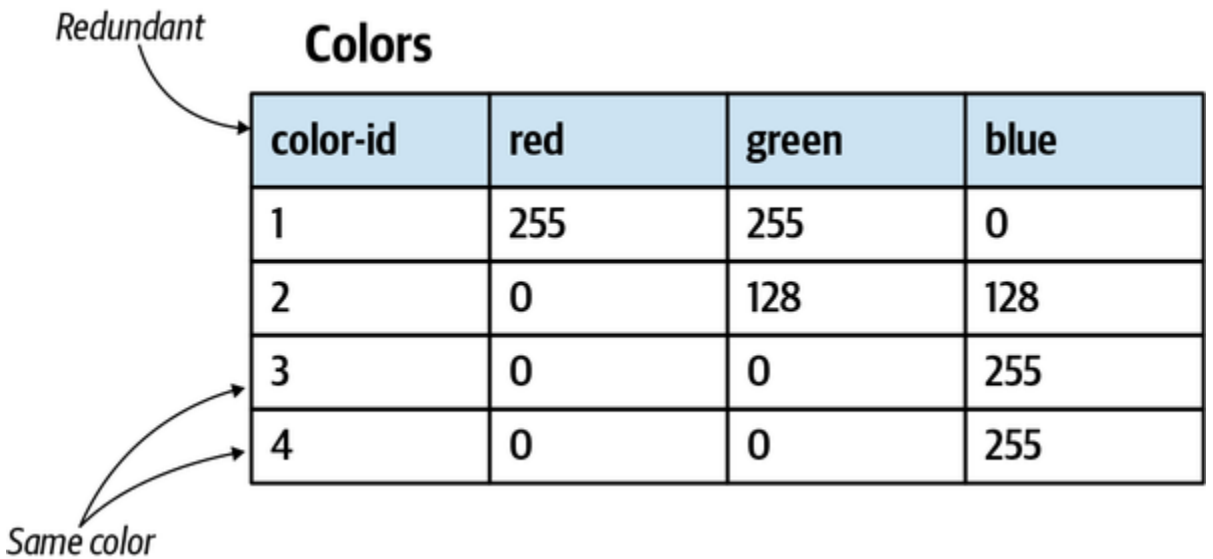
VALUE OBJECTS

- A value object is an object that can be identified by the composition of its values
- No identifier is necessary, like `id = 10` which would lead to confusion

```java
public class Color {
    private int red;
    private int green;
    private int blue;
}
```

IDENTIFIERS DOESN'T GO INTO VALUE OBJECTS

- Note that because identifier were introduced: `3`, and `4` are the same color
- This would not be considered a value object



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

VALUE OBJECTS AS JAVA RECORDS

- In Java, value objects can be simple one-liners.
- They are immutable, and readily have `toString`, `equals`, `hashCode`

```java
public record Color(int red, int green, int blue) {}
```

PRIMITIVE OBSESSION

Relying exclusively on the language's standard library's primitive data types—such as strings, integers, lists, or dictionaries — to represent concepts of the business domain is known as the primitive obsession code smell

EXAMPLE PRIMITIVE OBSESSION CODE SMELL

Java

```
class Person {
    private int    id;
    private String firstName;
    private String lastName;
    private String landlinePhone;
    private String mobilePhone;
    private String email;
    private int    heightMetric;
    private String countryCode;

    public Person(...) {...}
}
```

WHAT'S WRONG WITH PRIMITIVE OBSESSION?

- Consider `landLine` and `countryCode`, what are the valid numbers?
- How are you protecting those values?
- If there are multiple phone number types each validation may be duplicated.

WITHOUT PRIMITIVE OBSESSION

- The benefit is that any validation is done at the value object level
- Another benefit is that any method that is related belongs to the value object
- It adheres to the ubiquitous language

```java
class Person {
    private PersonId     id;
    private Name         name;
    private PhoneNumber  landline;
    private PhoneNumber  mobile;
    private EmailAddress email;
    private Height       height;
    private CountryCode  country;

    public Person(...) { ... }
}
```

VALUE OBJECTS CREATES RICH INTUITIVE METHODS

```java
var heightMetric = Height.metric(180);
var heightImperial = Height.imperial(5, 3);

var string1 = heightMetric.toString()                // "180cm"
var string2 = heightImperial.toString();             // "5 feet 3 inches"
var string3 = heightMetric.toImperial().toString(); // "5 feet 11 inches"

var firstIsHigher = heightMetric.isGreaterThan(heightImperial); // true
```

VALUE OBJECT AS MIXING COLORS

```java
var red = Color.fromRGB(255, 0, 0);
var green = Color.Green;
var yellow = red.mixWith(green);
var yellowString = yellow.toString();
```

IMPLEMENTATION AND IMMUTABILITY

- Value objects are *immutable*
- When a change is made to a value object, it typically should return *another object* with that change

EXAMPLE OF IMMUTABILITY WITH COLOR

```java
public record Color(int red, int green, int blue) {
    public Color {
        // Validate the input
    }

    public Color mixedWith(Color other) {
        return new Color(
            Math.min(this.red + other.red, 255),
            Math.min(this.green + other.green, 255),
            Math.min(this.blue + other.blue, 255)
        );
    }
}
```

WHEN TO USE VALUE OBJECTS

- Whenever you can
- Benefit is that they are free of side-effects and are thread safe
- Consider value objects for the domain's elements that describe properties of other objects
- Consider value objects for `firstName`, `lastName`, `expirationDate`, passwords, temperature, and (especially) money
- Look for libraries that have that pre-developed, for example the Squants project in Scala

LANGUAGE ARE OPTIMIZING FOR VALUE OBJECTS

- Language have or planning to optimize for having and using value objects
- Consider some of the advancements planned for Java

ENTITIES

- An entity is the opposite of a value object.
- It requires an explicit identification field to distinguish between the different instances of the entity.

VALUES OBJECTS DON'T HAVE IDENTITY

- In the following example, this is just a standard value object containing another value object
- As we all know people can have the same name, so how do we distinguish between two of these objects?

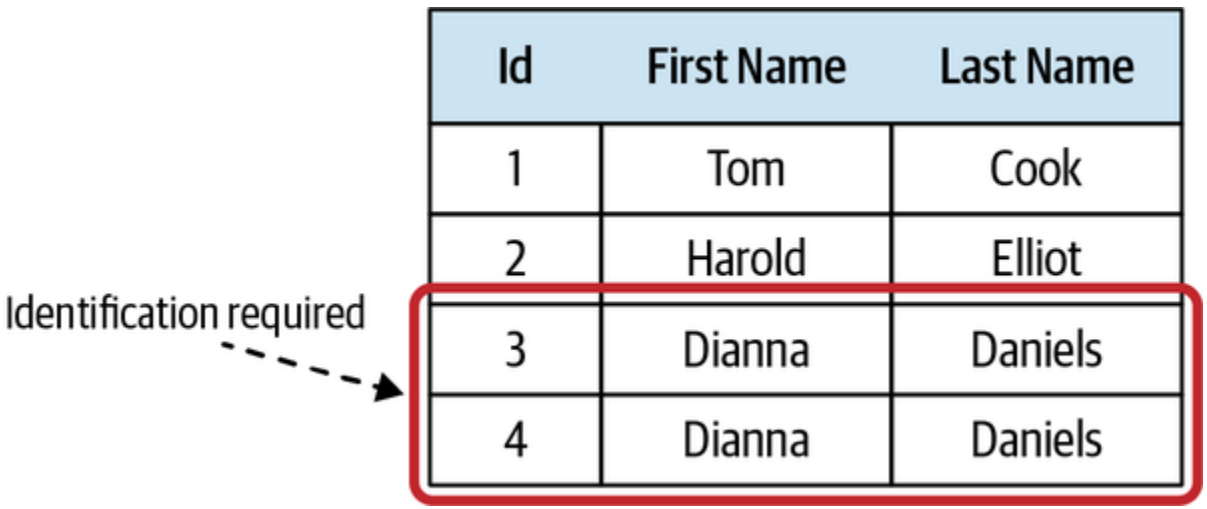
```java
public class Person {
    private Name name;
    public Person(Name name) {
        this.name = name;
    }
}
```

ADDING AN IDENTITY

- To distinguish between people we need an identifier: `EmployeeID`, `SocialSecurityNumber`, etc.
- The identifier field should be immutable since a person's identifier should remain constant, eventhough their name can change
- Identifiers must be unique

```java
public class Person {
    private final PersonId id;
    private Name name;
    public Person(PersonId id, Name name) {
        this.name = name;
    }
}
```

EXAMPLE OF TWO DISTINCT PEOPLE



ENTITIES ARE NOT IMMUTABLE

- Entities are not immutable, they are expected to change
- The value objects that constitute the entities are immutable
- Entities are not formally a part of the domain model, because they are building block for creating *aggregates*

AGGREGATES

- Aggregates are entities
- It requires explicit identification
- Draws a clear boundary between aggregate and it’s outer scope
- Aggregate’s logic has to validate all incoming modifications and ensure that the changes do not contradict its business rules

AGGREGATES PROTECT THE DATA

- Consistency of the data is enforced by the aggregate
- Processes or objects external to the aggregate are only allowed to read the aggregate's state
- State can only be mutated by executing corresponding methods on the aggregate's public interface

COMMANDING THE AGGREGATE

- State modifying methods exposed as the aggregate's public interface are referred to as *commands*
- You can implement commands by:
  - Using a public method
  - As a parameter object
  - Using the command pattern

ENFORCEMENT OF AGGREGATES

- An aggregate's public interface is responsible for validating the input and enforcing all the relevant business rules and invariants
- Strict boundary also ensures that all business logic related to the aggregate is implemented in one place: the aggregate itself

REVIEW OF LAYERS

DOMAIN LAYER

- Focuses purely on business logic and rules.
- Should be independent of the application and infrastructure layers.
- Contains entities, value objects, aggregates, domain services, and repository interfaces.

APPLICATION LAYER

- Interacts with the domain layer to execute use cases.
- Should be free from infrastructure concerns (e.g., no direct database access).
- Calls domain services, aggregates, and repositories to carry out business operations.

INFRASTRUCTURE LAYER

- Implements the technical details required to support the application and domain layers.
- Should implement interfaces defined in the domain layer for persistence, communication, etc.
- Manages cross-cutting concerns like logging, transactions, and security.

INTEGRATION WITH THE APPLICATION LAYER

- Integrating aggregates can then be done in the *application layer*
- In the following example, a repository represents the storage facility
- The consistency of the aggregate must be protected, transactional, and concurrent

```java
public ExecutionResult escalate(TicketId id, EscalationReason reason) {
  try {
      var ticket = ticketRepository.load(id);
      var cmd = new Escalate(reason);
      ticket.execute(cmd);
      ticketRepository.save(ticket);
      return ExecutionResult.Success;
  } catch (TransactionException ex) {
      return ExecutionResult.Error(ex.getMessage());
  }
}
```

TRANSACTION BOUNDARY

- The aggregate also acts as a transactional boundary
- Aggregate's state can only be modified by its own business logic, the aggregate also acts as a transactional boundary
- All changes to the aggregate's state should be committed transactionally as one atomic operation
- If an aggregate's state is modified, either all the changes are committed or none of them is

NO MULTI-AGGREGATE TRANSACTIONS

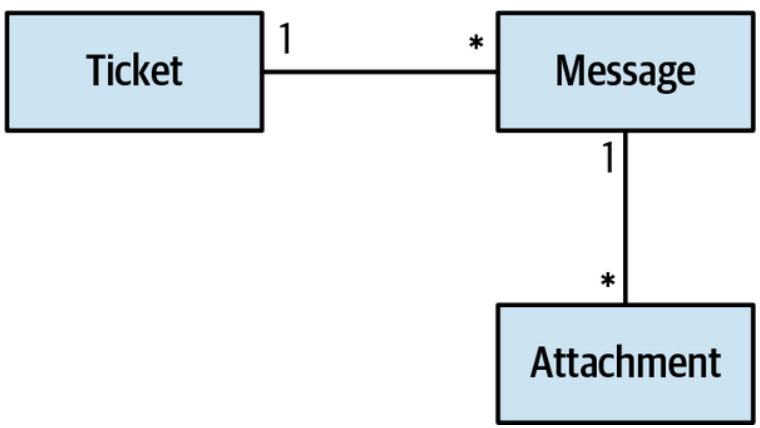
- No system operation can assume a multi-aggregate transaction.
- A change to an aggregate's state can only be committed individually, one aggregate per database transaction.

WHAT IF WE NEED TO HANDLE MULTIPLE ENTITIES?

- The one aggregate instance per transaction forces us to carefully design an aggregate's boundaries, ensuring that the design addresses the business domain's invariants and rules.
- The need to commit changes in multiple aggregates signals a wrong transaction boundary, and hence, wrong aggregate boundaries.
- What if we need to modify multiple objects in the same transaction?

HIERARCHY OF ENTITIES

- DDD prescribes that a system's design should be driven by its business domain
- To support changes to multiple objects that have to be applied in one atomic transaction, the aggregate pattern resembles a hierarchy of entities, all sharing transactional consistency



- The hierarchy contains both entities and value objects, and all of them belong to the same aggregate if they are bound by the domain's business logic

EXAMPLE OF HIERARCHY OF ENTITIES

Java

```
public class Ticket {
    private List<Message> messages;

    public void execute(evaluateAutomaticActions cmd) {
        if (this.isEscalated && this.remainingTimePercentage < 0.5 &&
            getUnreadMessagesCount(assignedAgent) > 0) {
            agent = AssignNewAgent();
        }
    }

    public int getUnreadMessagesCount(UserId id) {
        return messages.filter(x -> x.to == id && !x.wasRead).toList().size();
    }
}
```

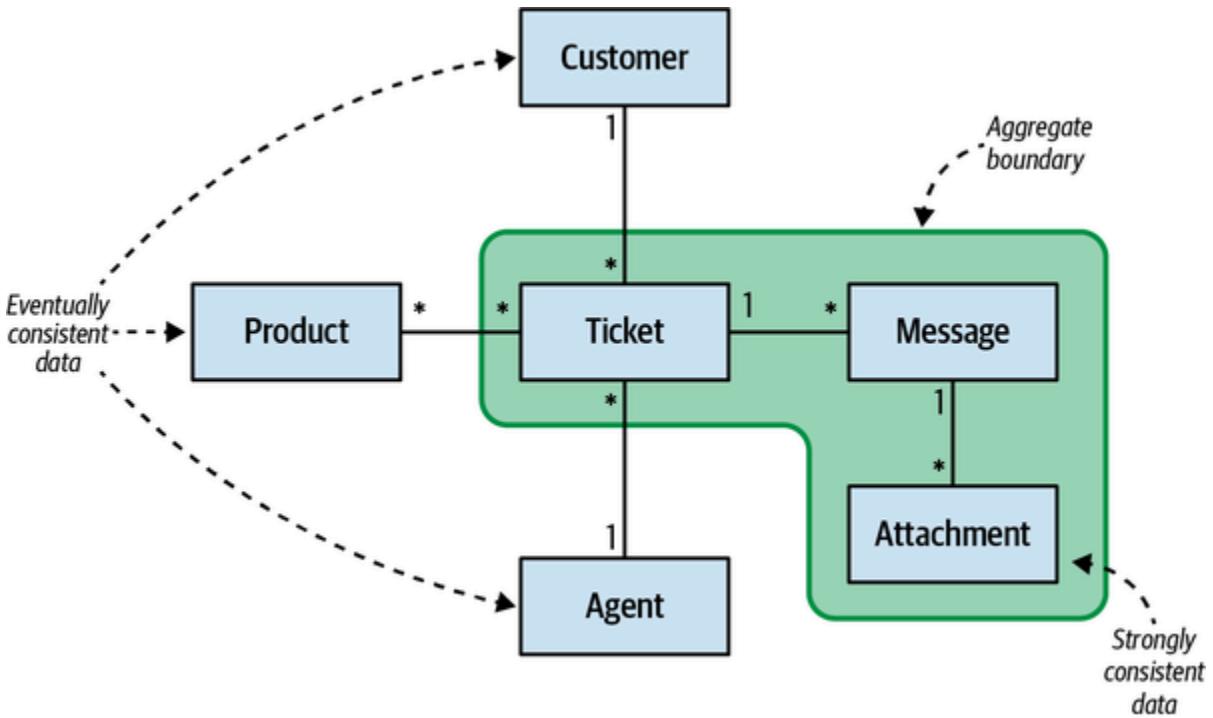
EXAMPLE OF HIERARCHY OF ENTITIES

- The method in the previous slide, checks the ticket's values to see whether it is escalated and whether the remaining processing time is less than the defined threshold of 50%
- It checks for messages that were not yet read by the current agent
- If all conditions are met, the ticket is requested to be reassigned to a different agent
- The aggregate ensures that all the conditions are checked against strongly consistent data

REFERENCING OTHER AGGREGATES

- Since all objects contained by an aggregate share the same transactional boundary, performance and scalability issues may arise if an aggregate grows too large.
- The consistency of the data can be a convenient guiding principle for designing an aggregate's boundaries.
- **Only the information that is required by the aggregate's business logic to be strongly consistent should be a part of the aggregate**

AGGREGATE AS CONSISTENCY BOUNDARY



KEEP AGGREGATES SMALL

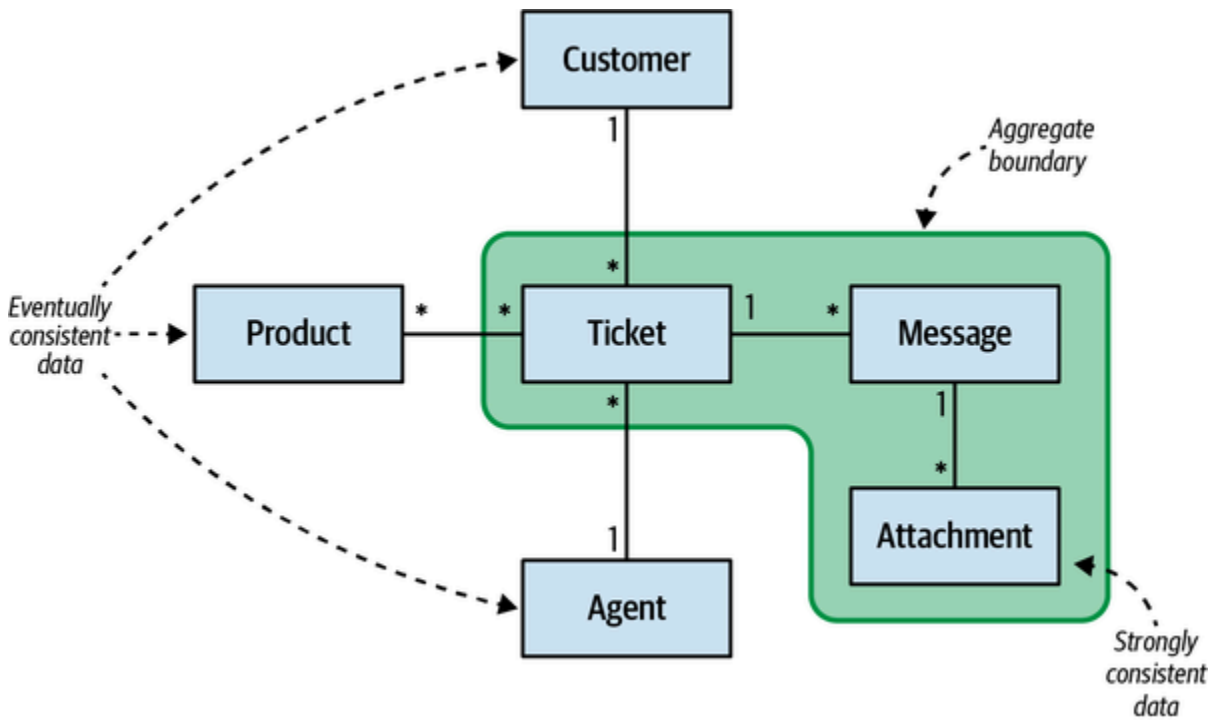
- Keep the aggregates as small as possible and include only objects that are required to be in a strongly consistent state by the aggregate's business logic
- In the following code example, notice that we are not interested in the entire `Product` nor the entire `User`, just their identifiers (entities)

```java
public class Ticket {
    private UserId          customer;
    private List<ProductId> products;
    private UserId          assignedAgent;
    private List<Message>   messages;
}
```

- **Reasoning behind referencing external aggregates by ID is to reify that these objects do not belong to the aggregate's boundary, and to ensure that each aggregate has its own transactional boundary.**

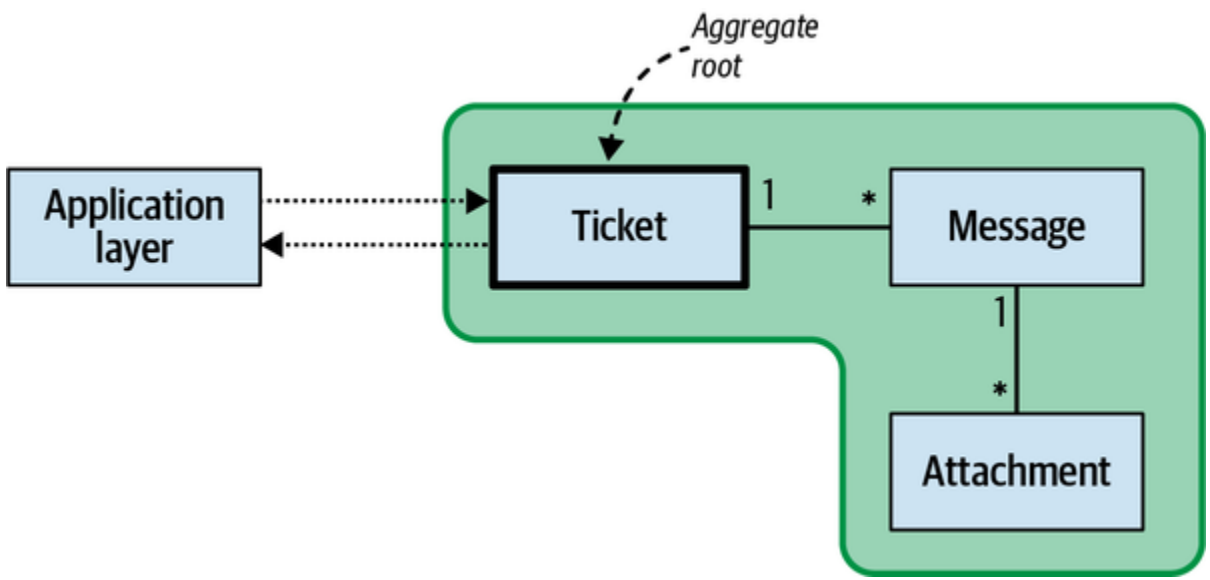
DECIDING WHAT TO AGGREGATE

To decide whether an entity belongs to an aggregate or not, examine whether the aggregate contains business logic that can lead to an invalid system state if *it will work on eventually consistent data*



AGGREGATE ROOT

- An aggregate's state can only be modified by executing one of its commands.
- Since an aggregate represents a hierarchy of entities, only one of them should be designated as the aggregate's public interface—*the aggregate root*



EXAMPLE OF AN AGGREGATE ROOT

Java

```
public class Ticket {
   private List<Message> _messages;

   public void execute(AcknowledgeMessage cmd) {
      var message =
        messages.stream().filter(x -> x.id == cmd.id)
        .findFirst().ifPresent(m -> m.wasRead = true);
   }
}
```

DOMAIN EVENTS

- A domain event is a message describing a significant event that has occurred in the business domain, for example:
  - Ticket assigned
  - Ticket escalated
  - Message received
- Domain events are *past tense*

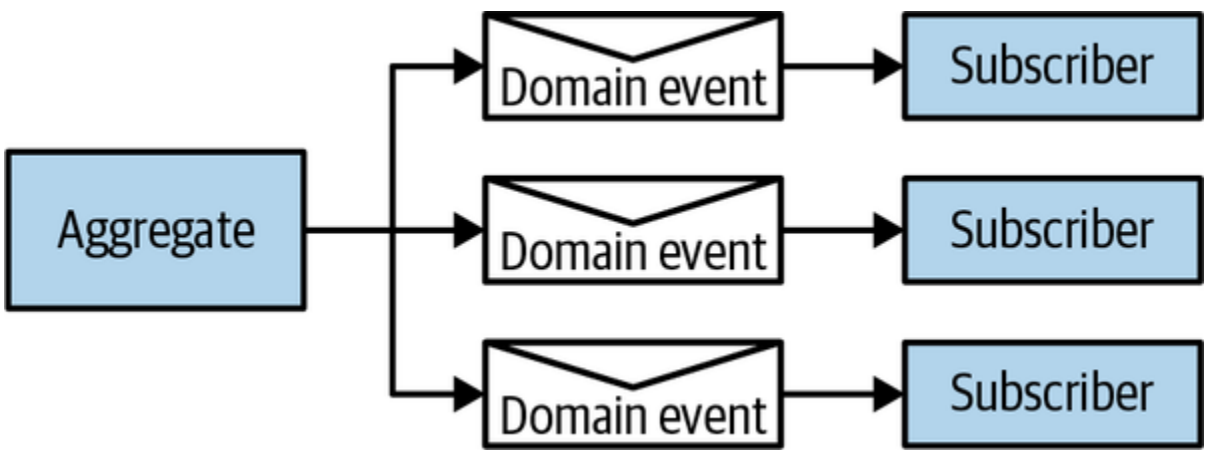
DOMAIN EVENT EXAMPLE

```Json
{
    "ticket-id": "c9d286ff-3bca-4f57-94d4-4d4e490867d1",
    "event-id": 146,
    "event-type": "ticket-escalated",
    "escalation-reason": "missed-sla",
    "escalation-time": 1628970815
}
```

- Ensure that domain events succinctly reflect exactly what has happened in the business domain

DOMAIN EVENTS PUBLIC INTERFACE

- Domain Events are part of the aggregate's public interface
- An aggregate publishes its domain events
- Other processes, aggregates, or even external systems can subscribe to and execute their own logic in response to the domain events



DOMAIN EVENT EXAMPLE

```java
public class Ticket {
    private List<DomainEvent> domainEvents;

    public void execute(RequestEscalation cmd)
    {
        if (!this.isEscalated && this.remainingTimePercentage <= 0) {
            this.isEscalated = true;
            var escalatedEvent = new TicketEscalated(id, cmd.Reason);
            domainEvents.append(escalatedEvent);
        }
    }
}
```

We will see how we can use communication software to manage those events

AGGREGATES MUST REFLECT THE UBIQUITOUS LANGUAGE

- Aggregates should reflect the ubiquitous language.
- The terminology that is used for the aggregate's name, its data members, its actions, and its domain events all should be formulated in the bounded context's ubiquitous language.
- As Eric Evans put it, the code must be based on the same language the developers use when they speak with one another and with domain experts. This is especially important for implementing complex business logic

DOMAIN SERVICES

- Business logic that doesn't belong to any aggregate or value object, or that seems to be relevant to multiple aggregates
- Stateless object that orchestrates calls to various components of the system to perform some calculation or analysis

DOMAIN SERVICE EXAMPLE

In the following example, calculation logic requires information from multiple sources

```java
public class ResponseTimeFrameCalculationService {
    public ResponseTimeframe calculateAgentResponseDeadline(UserId agentId,
        Priority priority, bool escalated, DateTime startTime) {
        var policy = departmentRepository.getDepartmentPolicy(agentId); (1)
        var maxProcTime = policy.getMaxResponseTimeFor(priority); (2)
        if (escalated) {
            maxProcTime = maxProcTime * policy.escalationFactor;
        }
        var shifts = departmentRepository.getUpcomingShifts(agentId,
            startTime, startTime.Add(policy.MaxAgentResponseTime)); (1)
        return CalculateTargetTime(maxProcTime, shifts);
    }
}
```

(1) The assigned agent's department

(2) The policy calculator

DOMAIN SERVICES ARE NOT MICROSERVICE

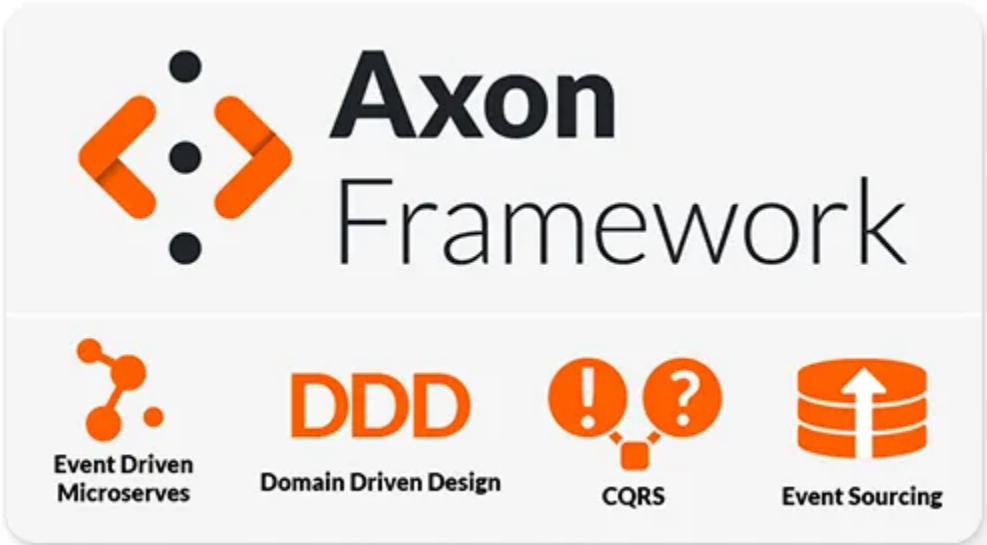
- Domain services have nothing to do with microservices, service-oriented architecture, or almost any other use of the word service in software engineering.
- It is just a stateless object used to host business logic.

FACTORIES

An object or method that implements object creation logic that’s too complex to be done directly by a constructor. It can also hide the concrete classes that are instantiated. A factory might be implemented as a static method of a class.

REPOSITORIES

An object that provides access to persistent entities and encapsulates the mechanism for accessing the database.



https://www.axoniq.io/products/axon-framework

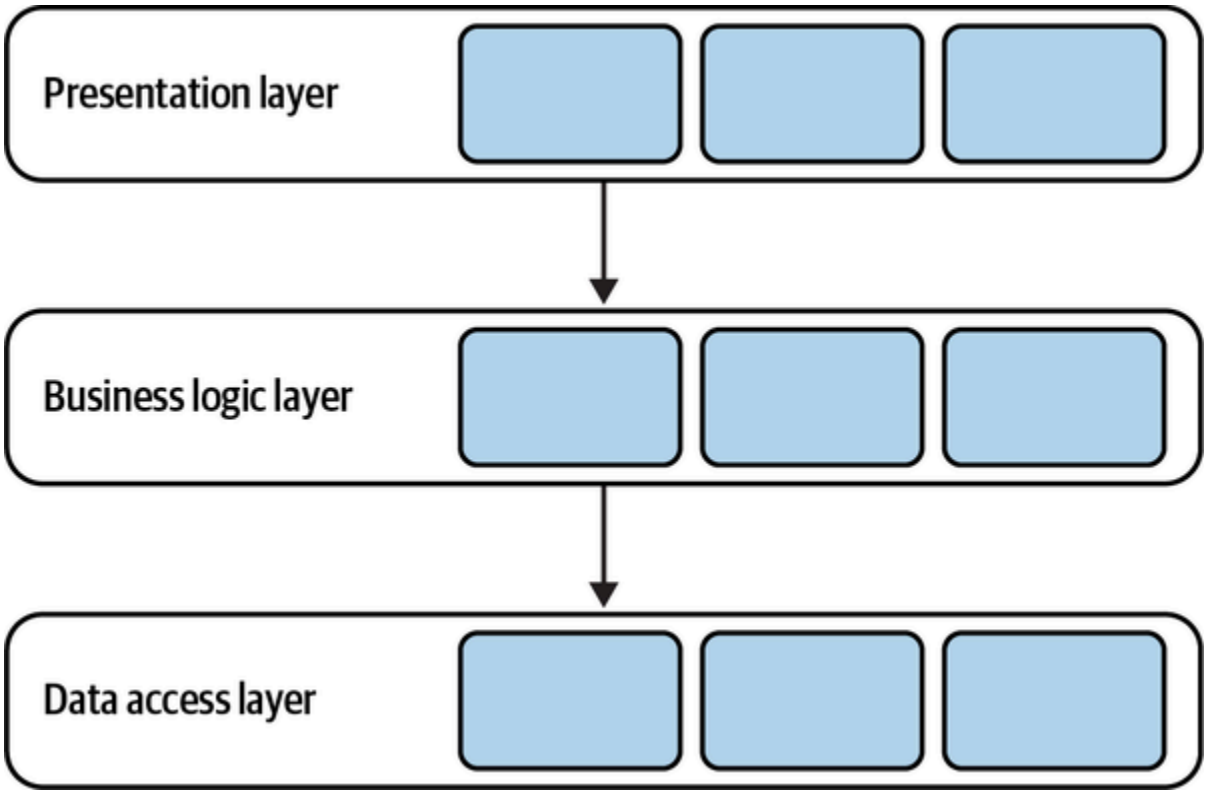

ARCHITECTURAL PATTERNS [TACTICAL]

CHOOSING THE APPROPRIATE ARCHITECTURE

- When the business logic has to change, it may not be evident what parts of the codebase have to be affected by the change.
- The change may have unexpected effects on seemingly unrelated parts of the system. Conversely, it may be easy to miss code that has to be modified.
- All of these issues dramatically increase the cost of maintaining the codebase.
- Architectural patterns introduce organizational principles for the different aspects of a codebase and present clear boundaries between them: how the business logic is wired to the system's input, output, and other infrastructural components
- Choosing the appropriate way to organize the codebase, or the correct architectural pattern, is crucial to support implementation of the business logic in the short term and alleviate maintenance in the long term

LAYERED ARCHITECTURE

LAYERED ARCHITECTURE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

LAYERED ARCHITECTURE

- One of the most common architectural pattersnm
- It organizes the codebase into horizontal layers, with each layer addressing one of the following technical concerns:
  - Interaction with the consumers (Presentation Layer - PL)
  - Implementing business logic (Business Layer - BL)
  - Persisting the data (Data Access Layer - DAL)

PRESENTATION LAYER

- program's user interface for interactions with its consumers
- layer denotes a graphical interface, such as a web interface or a desktop application
- All means for triggering behavior
  - Graphical user interface (GUI)
  - Command-line interface (CLI)
  - API for programmatic integration with other systems
  - Subscription to events in a message broker
  - Message topics for publishing outgoing events

Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

BUSINESS LOGIC LAYER

- Layer is responsible for implementing and encapsulating the program's business logic.
- This is the place where business decisions are implemented.
- "This layer is the heart of software" — Eric Evans
- Where business logic patterns are implemented—for example, active records or a domain model

Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

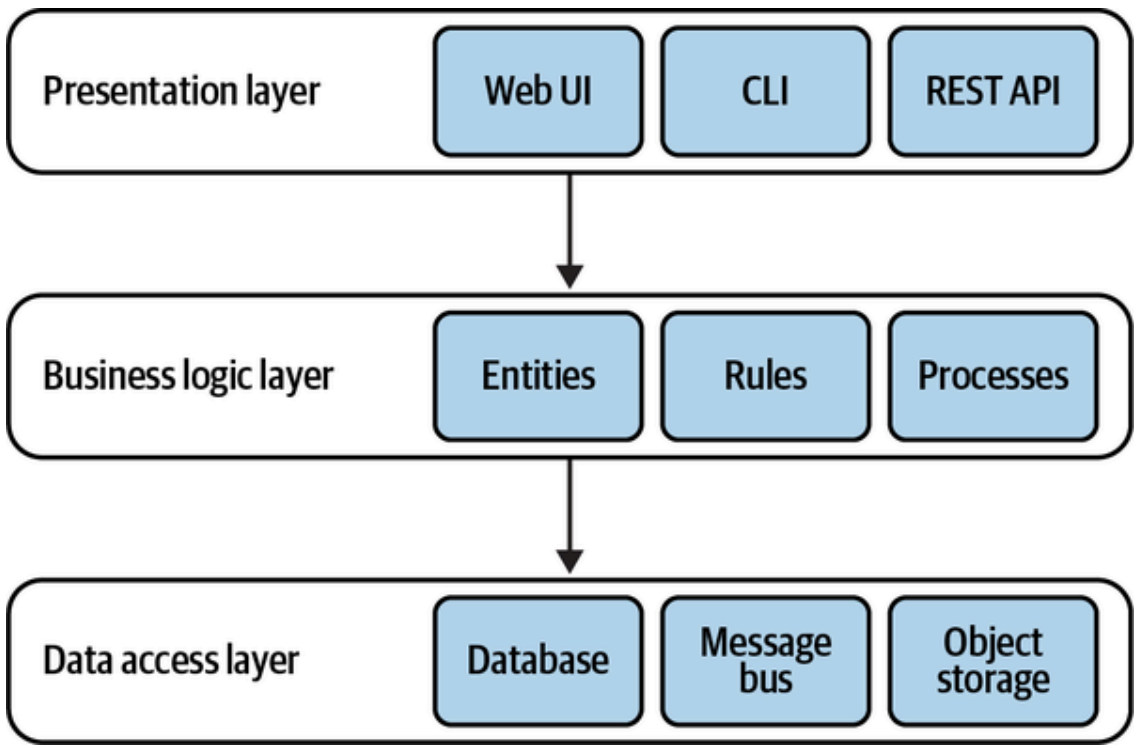
DATA ACCESS LAYER

- Persistence Mechanisms
- This include SQL datastores but so much more:
  - NoSQL
  - Cloud based storage
  - Messaging systems
  - External APIs (language translation, stock price services, etc)

Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

COMMUNICATION BETWEEN LAYERS

- Each layer can hold a dependency only on the layer directly beneath it
- This enforces decoupling of implementation concerns and reduces the knowledge shared between the layers

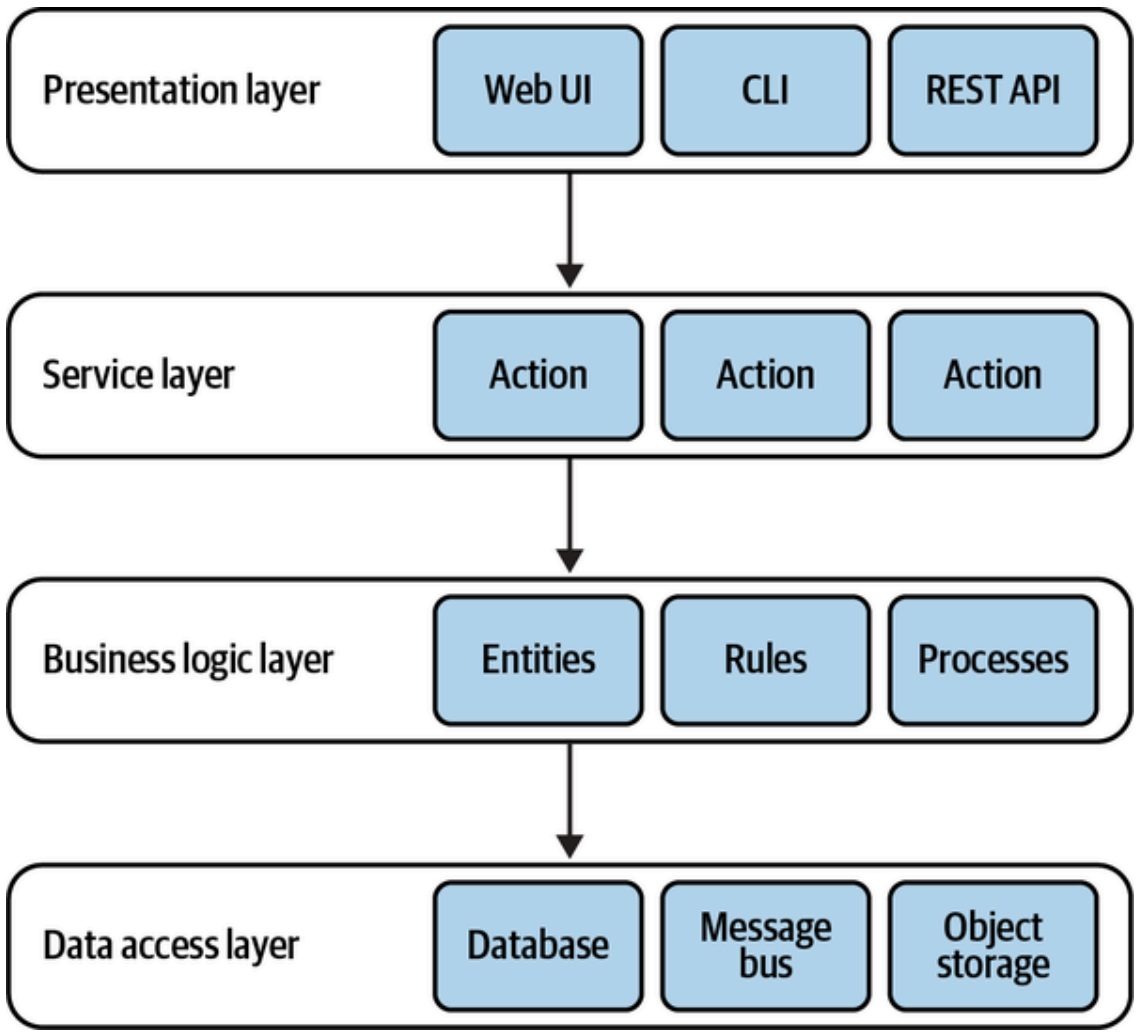


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

ADDED A SERVICE LAYER

- It’s common to see the layered architecture pattern extended with an additional layer: the service layer
- The service layer acts as an intermediary between the program’s presentation and business logic layers.

LAYERED ARCHITECTURE WITH SERVICE LAYER



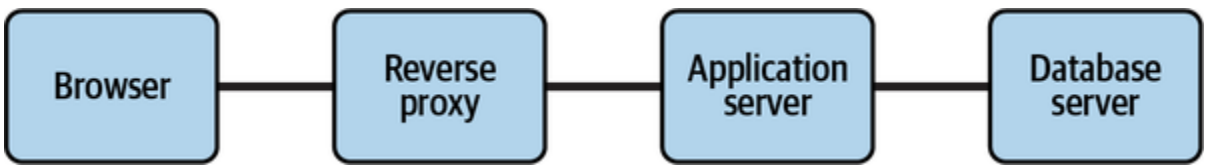
Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

WHEN TO USE LAYERED ARCHITECTURE

- Good fit for a system with its business logic implemented using the transaction script or active record pattern
- Bad fit to implement a domain model. In a domain model, the business entities (aggregates and value objects) should have no dependency and no knowledge of the underlying infrastructure

TIERS VS LAYERS

- The layers architecture is often confused with the N-Tier architecture, and vice versa
- They are different: A layer is a logical boundary, whereas a tier is a physical boundary
- Layers are bound by the same lifecycle: they are implemented, evolved, and deployed as one single unit
- Tiers is an independently deployable service, server, or system.



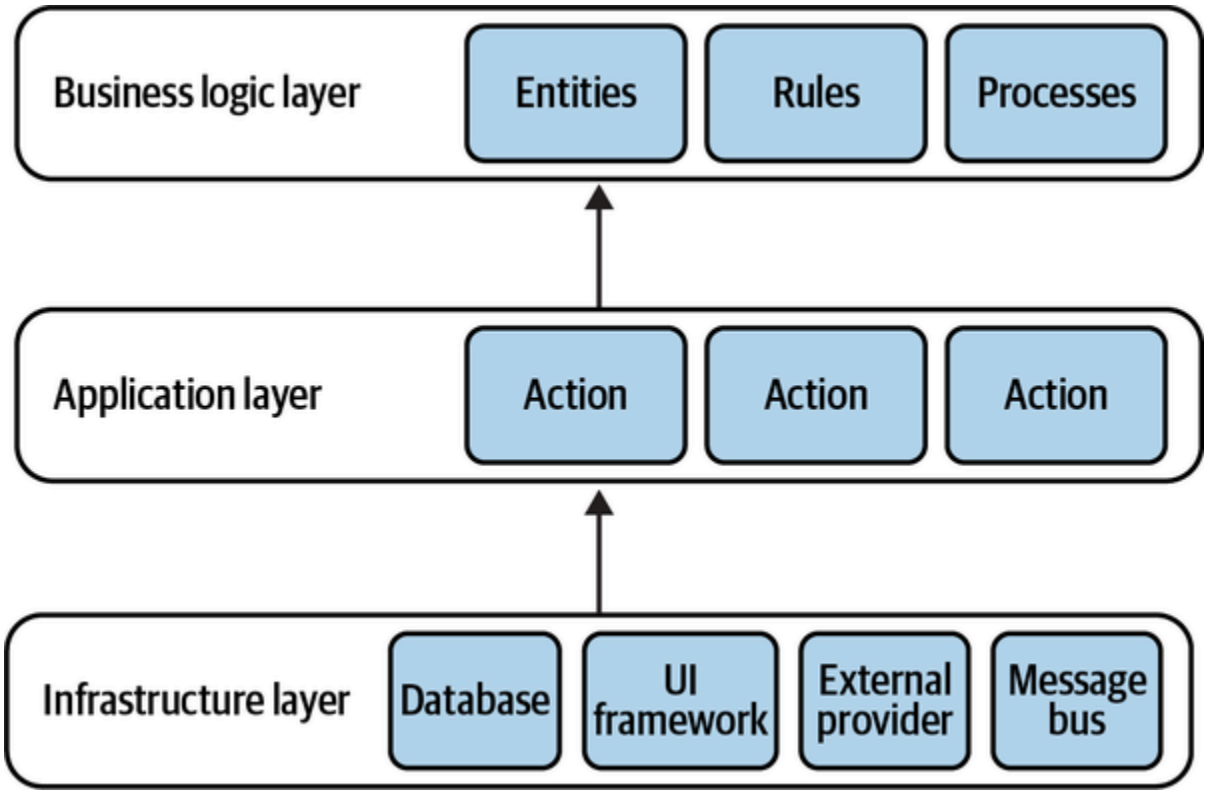
Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

PORTS & ADAPTERS

PORTS & ADAPTERS

- Architecture that addresses the shortcomings of the layered architecture and is a better fit for implementation of more complex business logic
- The dependency inversion principle (DIP) states that high-level modules, which implement the business logic, should not depend on low-level modules
- Notice the arrows for dependencies. The business logic is used and doesn't have any idea about the application it is housed in, nor the infrastructure

PORTS & ADAPTERS

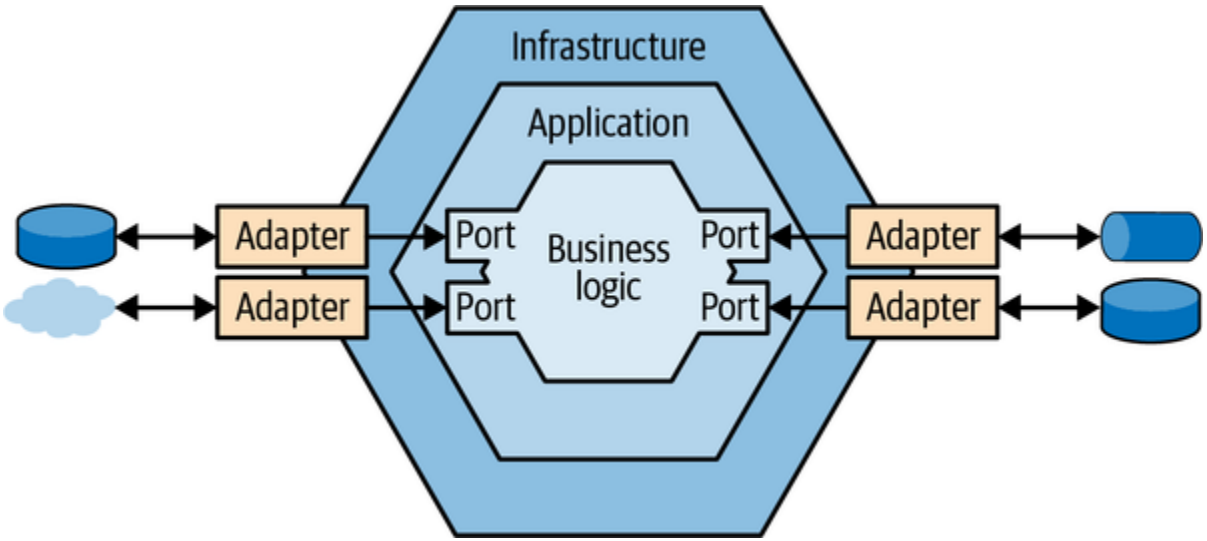


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

INTEGRATION WITH PORTS AND ADAPTERS

- Instead of referencing and calling the infrastructural components directly, the business logic layer defines “ports” that have to be implemented by the infrastructure layer.
- The infrastructure layer implements “adapters”: concrete implementations of the ports’ interfaces for working with different technologies

PORTS AND ADAPTERS ARCHITECTURE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

WHEN TO USE PORTS & ADAPTERS?

The decoupling of the business logic from all technological concerns makes the ports & adapters architecture a perfect fit for business logic implemented with the domain model pattern

A NOTE ABOUT JAVA MODULES

- Using modules in java for contexts or dividing layers is a strong possibility
- One of the benefits for doing so is that you can define strictly what can be accessed and what can't
- You can also define who has access to what java packages/modules

https://www.baeldung.com/java-modules-ddd-bounded-contexts

CQRS (COMMAND QUERY RESPONSIBILITY SEGREGATION)

CQRS (COMMAND QUERY RESPONSIBILITY SEGREGATION)

- Provides the possibility of materializing projected models into physical databases that can be used for flexible querying options
- Pattern segregates the responsibilities of the system's models.
- There are two types of models:
  - The command execution model
  - The read models

COMMAND EXECUTION MODEL

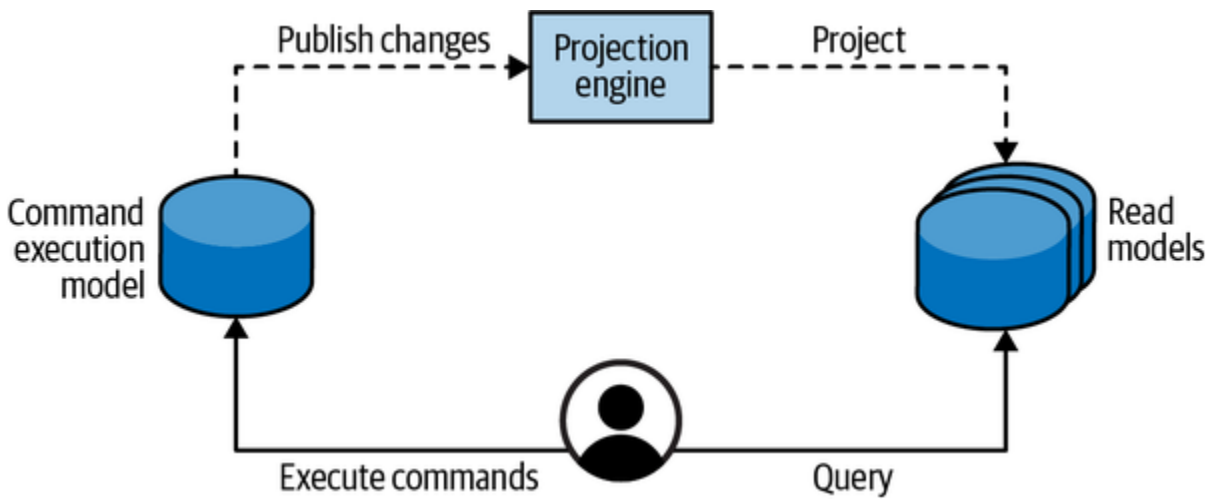
- CQRS devotes a single model to executing operations that modify the system's state (system commands).
- This model is used to implement the business logic, validate rules, and enforce invariants.
- The command execution model is also the only model representing strongly consistent data—the system's source of truth.
- It should be possible to read the strongly consistent state of a business entity and have optimistic concurrency support when updating it.

READ MODELS (PROJECTIONS)

- The system can define as many models as needed to present data to users or supply information to other systems.
- A read model is a precached projection.
- It can reside in a durable database, flat file, or in-memory cache. Proper implementation of CQRS allows for wiping out all data of a projection and regenerating it from scratch.
- This also enables extending the system with additional projections in the future—models that couldn't have been foreseen originally.
- Read models are read-only.
- **None of the system's operations can directly modify the read models' data.**

PROJECTING READ MODELS

- For the read models to work, the system has to project changes from the command execution model to all its read models.
- The projection of read models is similar to the notion of a materialized view in relational databases: whenever source tables are updated, the changes have to be reflected in the precached views.

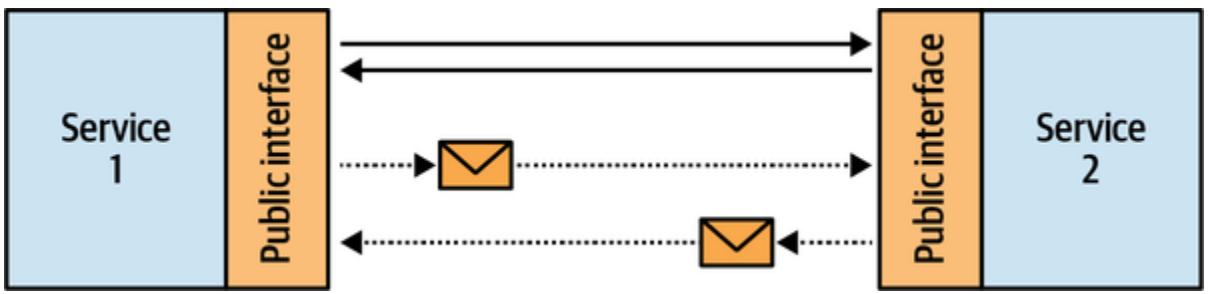


APPLICATION


MICROSERVICES [APPLICATION]

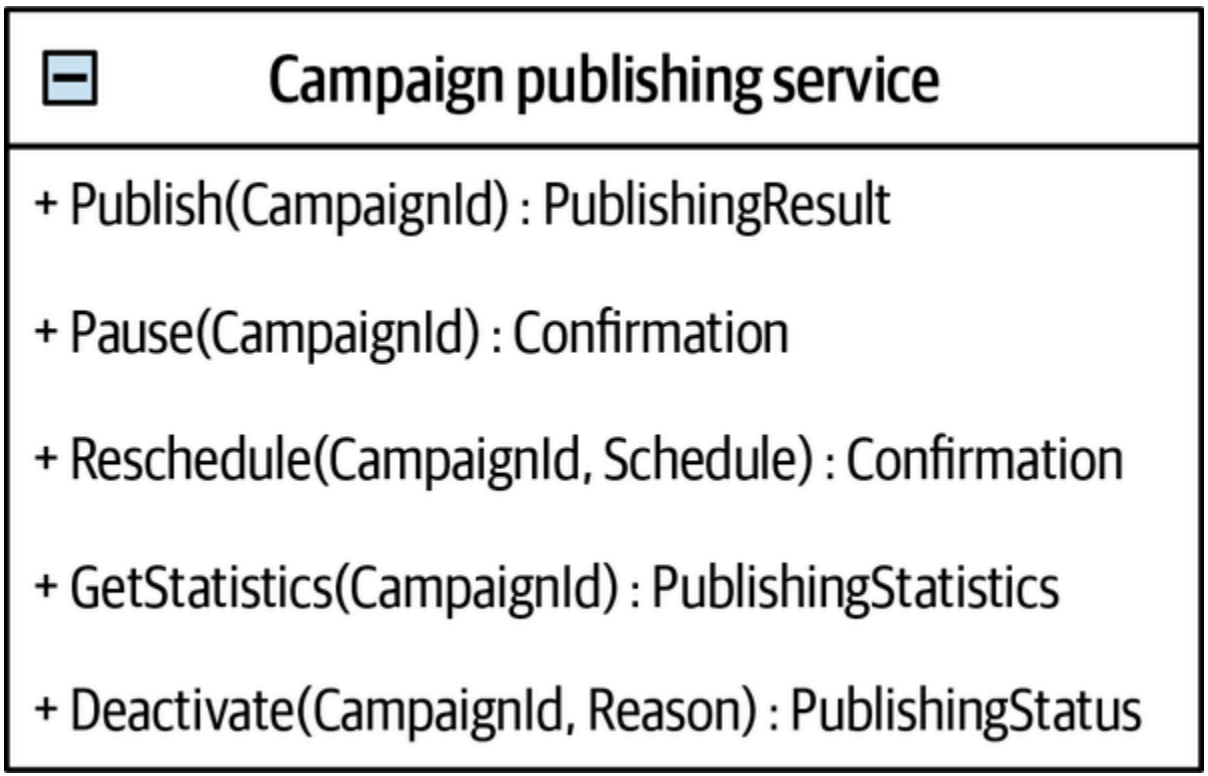
SERVICE

- A service is a mechanism that enables access to one or more capabilities, where the access is provided using a prescribed interface
- The prescribed interface is any mechanism for getting data in or out of a service.
- It can be synchronous, such as a request/response model, or asynchronous, such as a model that is producing and consuming events.



DESCRIBING A SERVICE

A well-expressed interface is enough to describe the functionality implemented by a service



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

MICROSERVICE

- Smaller scale of a service
- Smaller interface makes it easier to understand both the function of a single service and its integration with other system components.
- Reducing a service's functionality also limits its reasons for change and makes the service more autonomous for development, management, and scale

MICROSERVICE ENCAPSULATES DATABASES

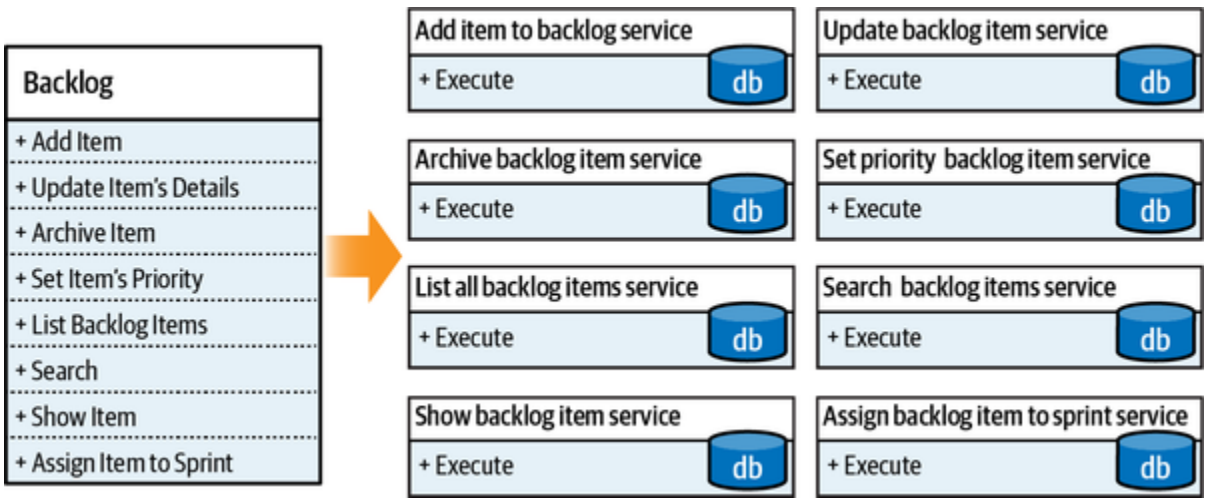
- Hence, microservices encapsulate their databases.
- The data can only be accessed through a much more compact, integration-oriented public interface.

DECOMPOSITION TO MICROSERVICES

- Decomposing is taking large service and decomposing into more fine-grained services
- Decomposing is difficult, but you can use business cases, domains, and bounded contexts to guide how to decompose

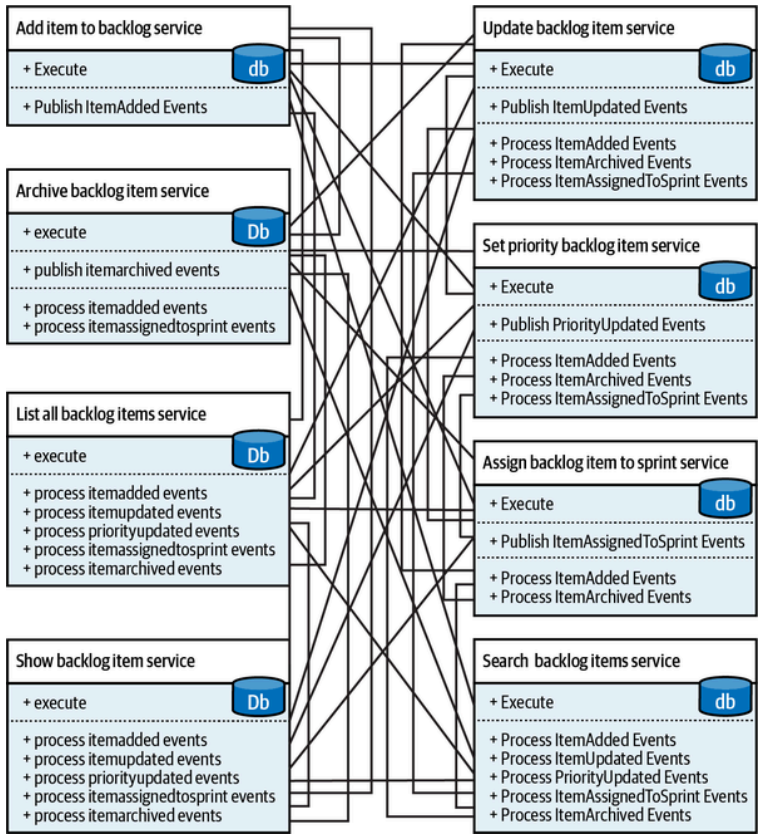
TOO SMALL DECOMPOSITION

Given the following decomposition, what would the end result be?



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

THE RESULT OF GOING TOO SMALL



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

THE PROBLEM WITH TOO SMALL MICROSERVICES

- You would need to offer an exposition of "database" calls since databases are not exposed
- That exposition causes many "backdoors" to implement

MICROSERVICE GOALS

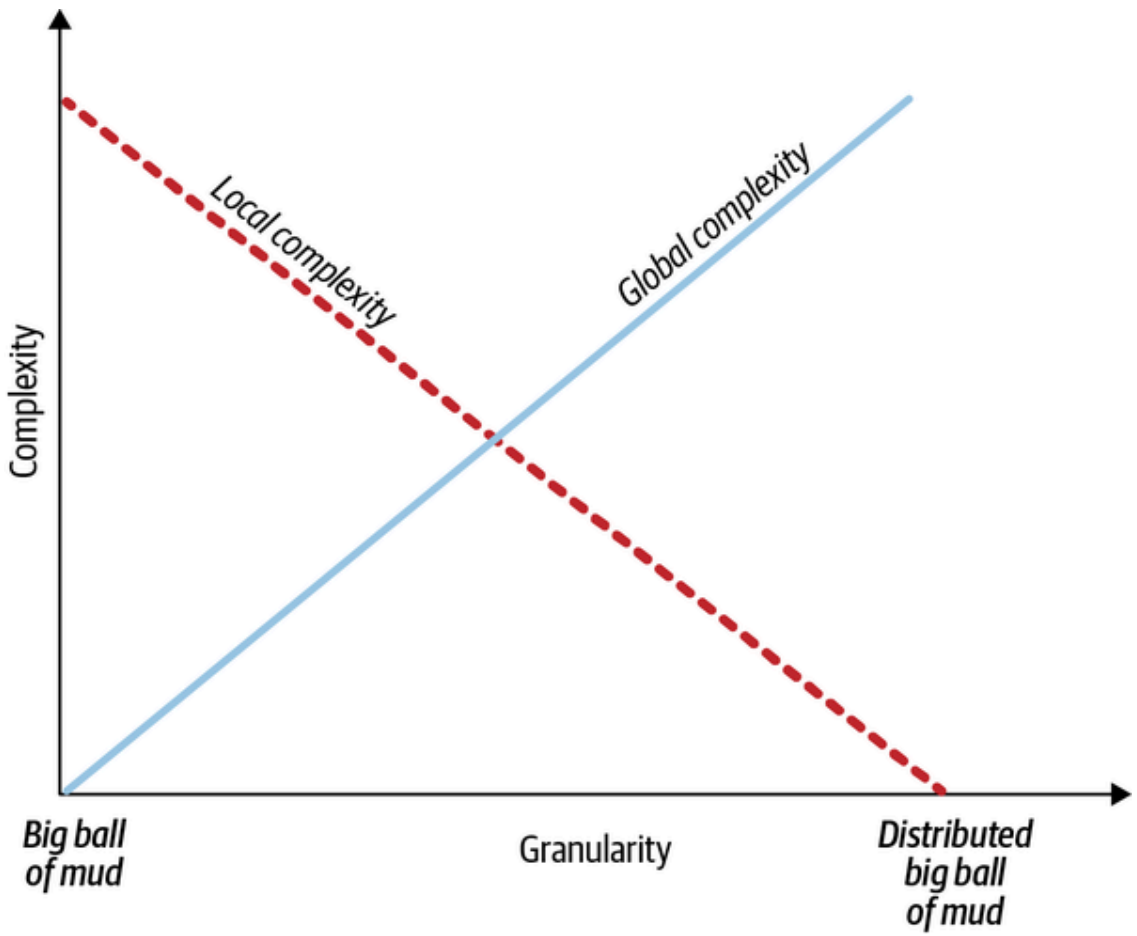
- With the big ball of mud: Each service ended up being much simpler than the original design, however the resultant system became orders of magnitude more complex.
- The goal of the microservices architecture is to produce a flexible system.
- In a proper microservices-based system, however decoupled, the services still have to be integrated and communicate with each other.

LOCAL VS. GLOBAL COMPLEXITY

- *Local complexity* is the complexity of each individual microservice
- *Global complexity* is the complexity of the whole system.

LOCAL VS. GLOBAL TRADEOFF

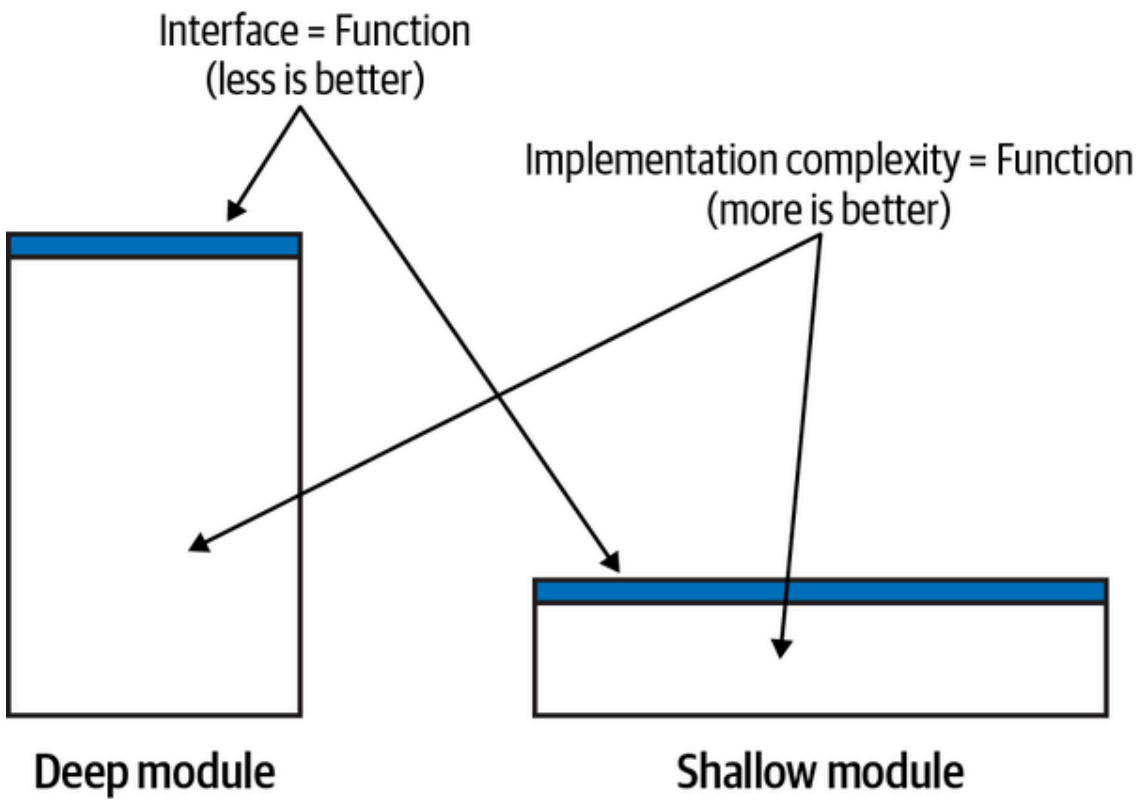
To design a proper microservices-based system, we have to optimize both global and local complexities. Setting the design goal of optimizing either one individually is a local optima. The global optima balances both complexities



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

DEEP MICROSERVICES

- A *function* is what the module is supposed to do—its business functionality.
- The *logic* is the module's business logic—how the module implements its business functionality

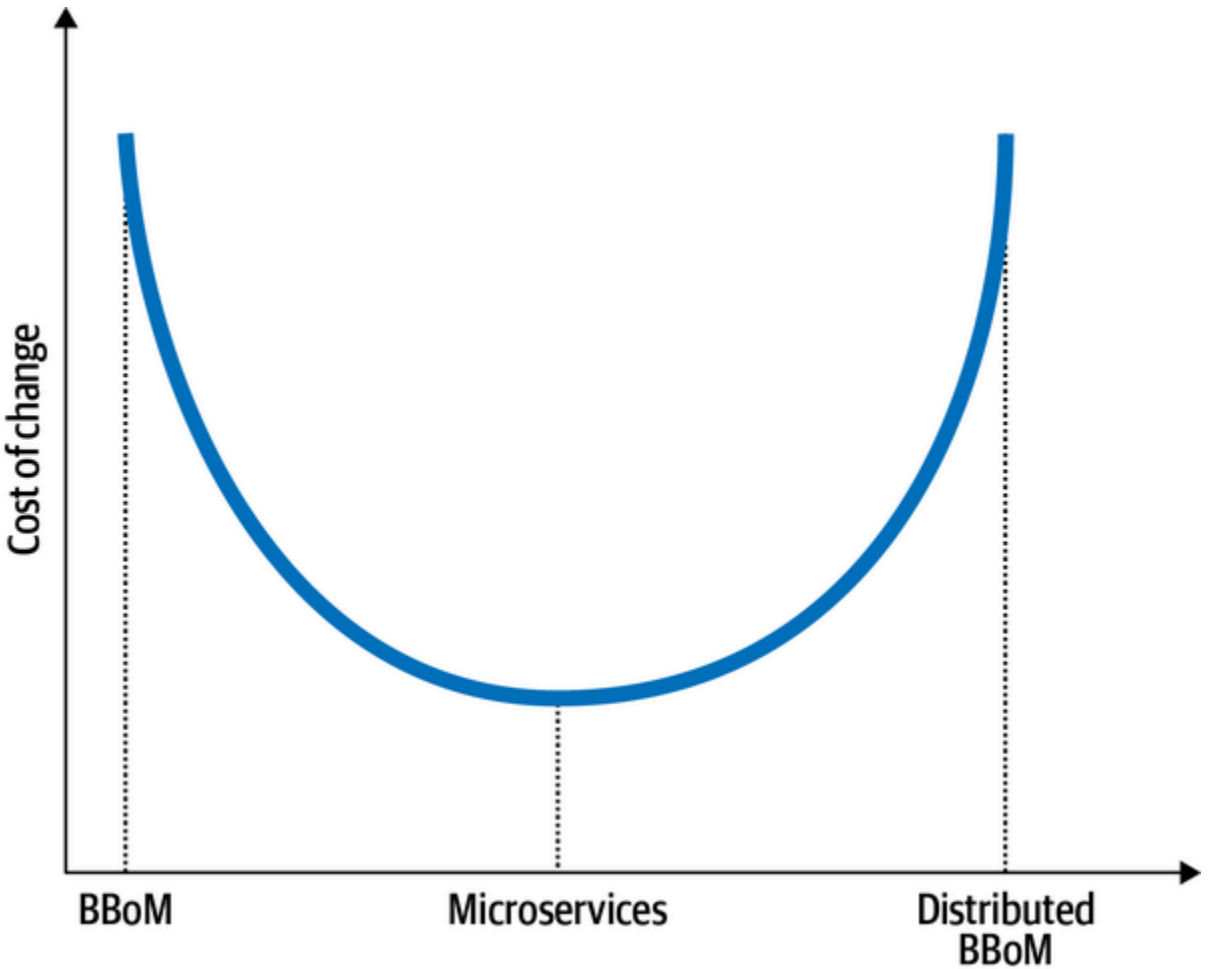


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

MICROSERVICES AND MODULARITY

- Shallow services are also the reason why so many microservices-oriented projects fail
- If we decompose a monolith into services, the cost of introducing a change goes down. It is minimized when the system is decomposed into microservices
- However, if you keep decomposing past the microservices threshold, the deep services will become more and more shallow. Their interfaces will grow back up
- This time, due to integration needs, the cost of introducing a change will go up as well, and the overall system's architecture will turn into the dreaded distributed big ball of mud

COST OF CHANGE



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

DOMAIN DRIVEN DESIGN AND MICROSERVICES

DIFFERENT BOUNDED CONTEXTS

- The different decompositions to bounded contexts attribute different requirements, such as different teams' sizes and structures, lifecycle dependencies, and so on.
- **But can we say that all the valid bounded contexts in this example are necessarily microservices? No**
- Especially considering the relatively wide functionalities of the two bounded contexts in decomposition

BOUNDED CONTEXTS AND MICROSERVICES?

- Therefore, the relationship between microservices and bounded contexts is not symmetric.
- **Although microservices are bounded contexts, not every bounded context is a microservice**
- Bounded contexts, on the other hand, denote the boundaries of the largest valid monolith (not be confused with big ball of mud)

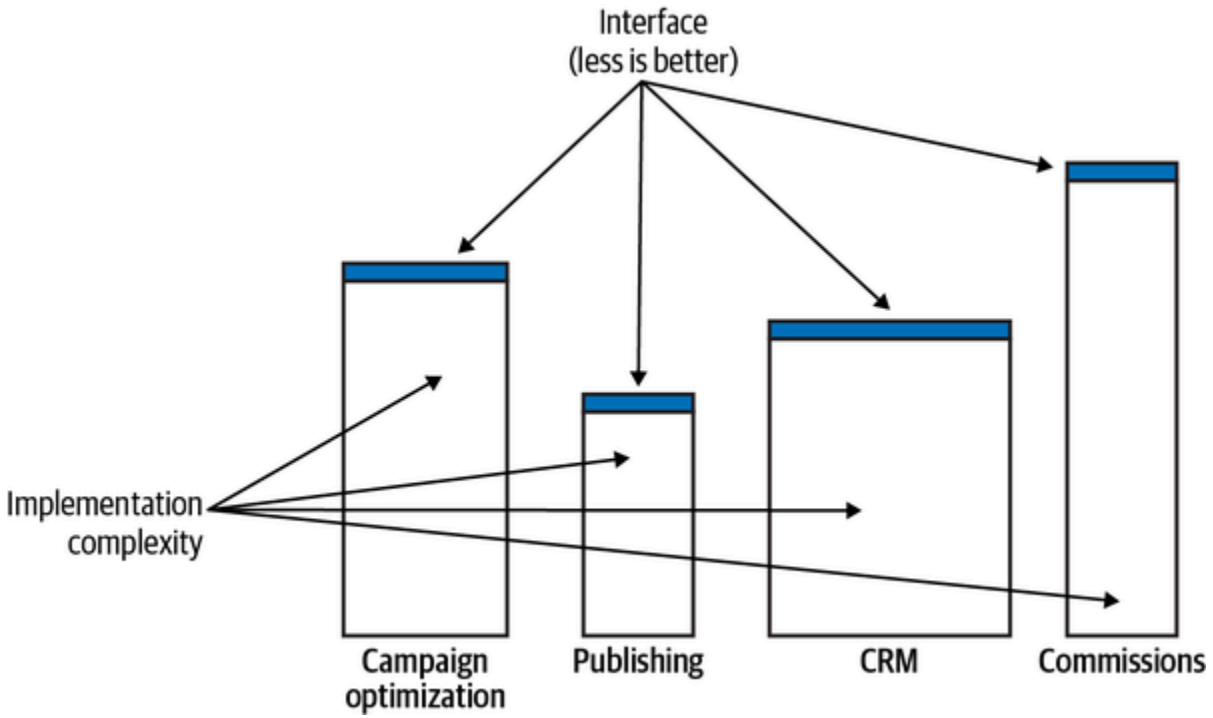
ARE AGGREGATES BETTER FOR MICROSERVICES?

- While bounded contexts impose limits on the widest valid boundaries, the aggregate pattern does the opposite.
- The aggregate's boundary is the narrowest boundary possible.
- Decomposing an aggregate into multiple physical services, or bounded contexts, is not only suboptimal but, leads to undesired consequences

ARE SUBDOMAINS BETTER FOR MICROSERVICES?

- A more balanced heuristic for designing microservices is to align the services with the boundaries of business subdomains
- Subdomains are correlated with fine-grained business capabilities.
- From a technical standpoint, subdomains represent sets of coherent use cases: using the same model of the business domain, working on the same or closely related data, and having a strong functional relationship.
- The subdomains' granularity and the focus on the functionality—the "what" rather than the "how"—makes subdomains naturally deep modules.

SUBDOMAINS AND MICROSERVICES



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.


EVENT STORMING [STRATEGIC]

WHAT IS EVENT STORMING?

- Low-tech activity for a group of people to brainstorm and rapidly model a business process.
- In a sense, EventStorming is a tactical tool for sharing business domain knowledge.
- An EventStorming session has a scope: the business process that the group is interested in exploring.
- The participants are exploring the process as a series of domain events, represented by sticky notes, over a timeline. Step by step, the model is enhanced with additional concepts—actors, commands, external systems, and others—until all of its elements tell the story of how the business process works.

Just keep in mind that the goal of the workshop is to learn as much as possible in the shortest time possible. We invite key people to the workshop, and we don't want to waste their valuable time.

— Alberto Brandolini: Creator of the EventStorming workshop

WHY?

- When new domain knowledge is discovered, it should be leveraged to evolve the design and make it more resilient.
- Unfortunately, changes in domain knowledge are not always positive: domain knowledge can be lost. As time goes by, documentation often becomes stale, people who were working on the original design leave the company, and new functionality is added in an ad hoc manner until, at one point, the codebase gains the dubious status of a legacy system.
- It’s vital to prevent such degradation of domain knowledge proactively.
- An effective tool for recovering domain knowledge is the EventStorming workshop

PEOPLE

- A diverse group of people should participate in the workshop.
- Anyone related to the business domain in question can participate: engineers, domain experts, product owners, testers, UI/UX designers, support personnel, and so on.
- As more people with different backgrounds are involved, more knowledge will be discovered.
- Take care not to make the group too big, however.
- Every participant should be able to contribute to the process, but this can be challenging for groups of more than 10 participants.

REQUIREMENTS

Modeling space

First, you need a large modeling space. A whole wall covered with butcher paper makes the best modeling space A large whiteboard can fit the purpose as well, but it has to be as big as possible—you will need all the modeling space you can get.

Sticky notes

Next, you need lots of sticky notes of different colors. The notes will be used to represent different concepts of the business domain, and every participant should be able to add them freely, so make sure you have enough colors and enough for everyone. The colors that are traditionally used for EventStorming are described in the next section. It's best to stick to these conventions, if possible, to be consistent with all of the currently available EventStorming books and trainings.

REQUIREMENTS

Markers

You'll also need markers that you can use to write on the sticky notes. Again, supplies shouldn't be a bottleneck for knowledge sharing—there should be enough markers for all participants.

Snacks

A typical EventStorming session lasts about two to four hours, so bring some healthy snacks for energy replenishment.

Room

Finally, you need a spacious room. Ensure there isn't a huge table in the middle that will prevent participants from moving freely and observing the modeling space. Also, chairs are a big no-no for EventStorming sessions. You want people to participate and share knowledge, not sit in a corner and zone out. Therefore, if possible, take the chairs out of the room.

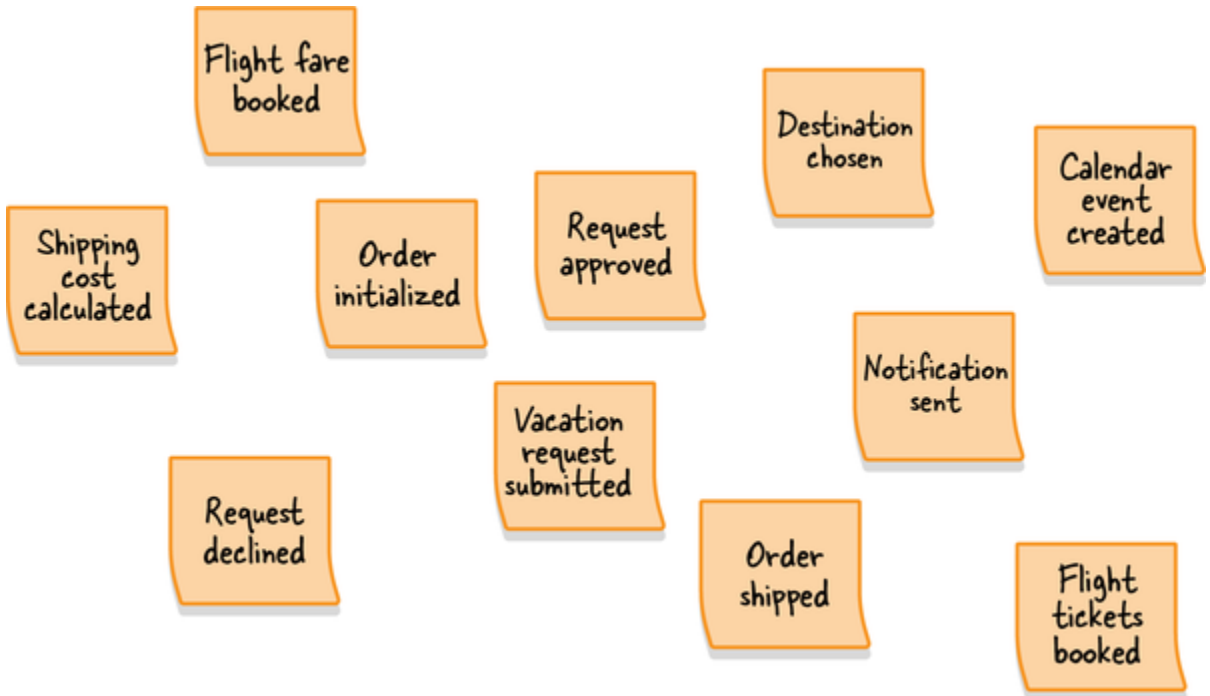
STEP 1: UNSTRUCTURED EXPLORATION

- EventStorming starts with a brainstorm of the domain events related to the business domain being explored.
- A domain event is something interesting that has happened in the business.
- It's important to formulate domain events in the *past tense*—they are describing things that have already happened.

STEP 1: UNSTRUCTURED EXPLORATION (CONTINUED)

- All participants are grabbing a bunch of orange sticky notes, writing down whatever domain events come to mind, and sticking them to the modeling surface.
- There is no need to worry about ordering events, or even about redundancy
- This step is all about brainstorming the possible things that can happen in the business domain
- The group should continue generating domain events until the rate of adding new ones slows significantly.

STEP 1: UNSTRUCTURED EXPLORATION (CONTINUED)

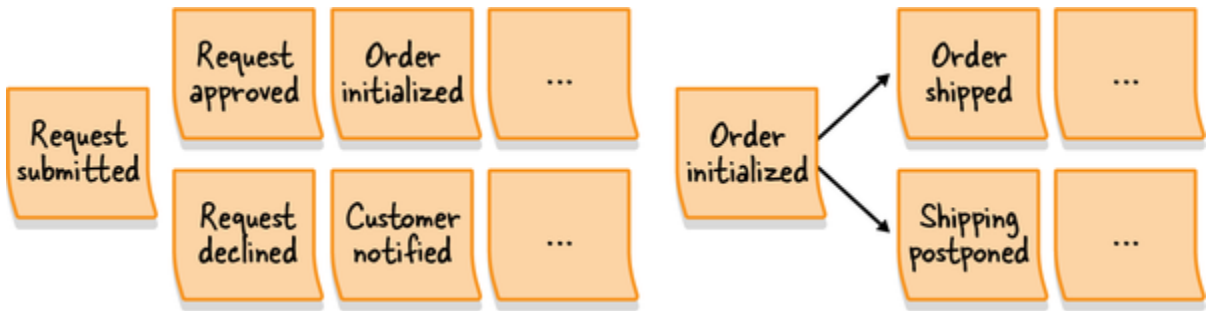


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 2: TIMELINES

- Next, the participants go over the generated domain events and organize them in the order in which they occur in the business domain.
- The events should start with the “happy path scenario”: the flow that describes a successful business scenario.
- Once the “happy path” is done, alternative scenarios can be added—for example, paths where errors are encountered or different business decisions are taken.
- The flow branching can be expressed as two flows coming from the preceding event or with arrows drawn on the modeling surface
- This step is also the time to fix incorrect events, remove duplicates, and of course, add missing events if necessary.

STEP 2: TIMELINES (CONTINUED)

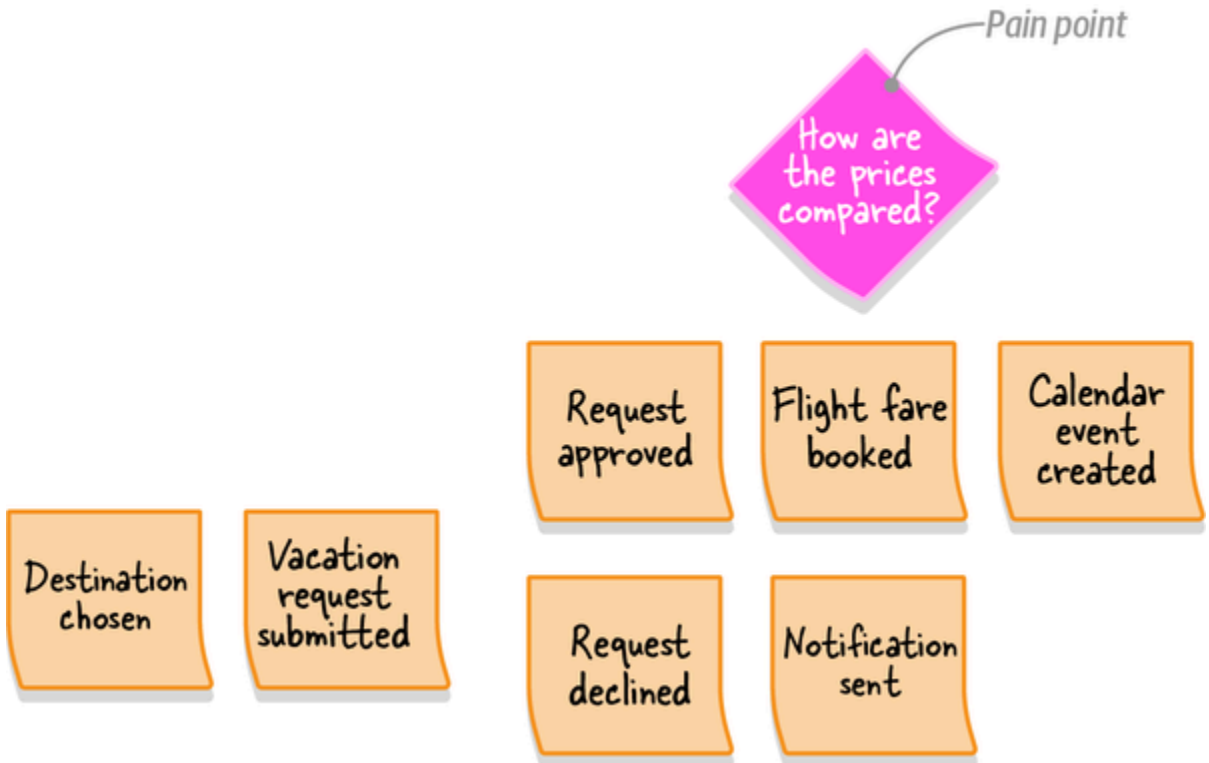


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 3: PAIN POINTS

- Once you have the events organized in a timeline, use this broad view to identify points in the process that require attention.
- These can be bottlenecks, manual steps that require automation, missing documentation, or missing domain knowledge.
- It’s important to make these inefficiencies explicit so that it will be easy to return to them as the EventStorming session progresses, or to address them afterward.
- The pain points are marked with rotated (diamond) pink sticky notes
- As a facilitator, be aware of the participants’ comments throughout the process. When an issue or a concern is raised, document it as a pain point

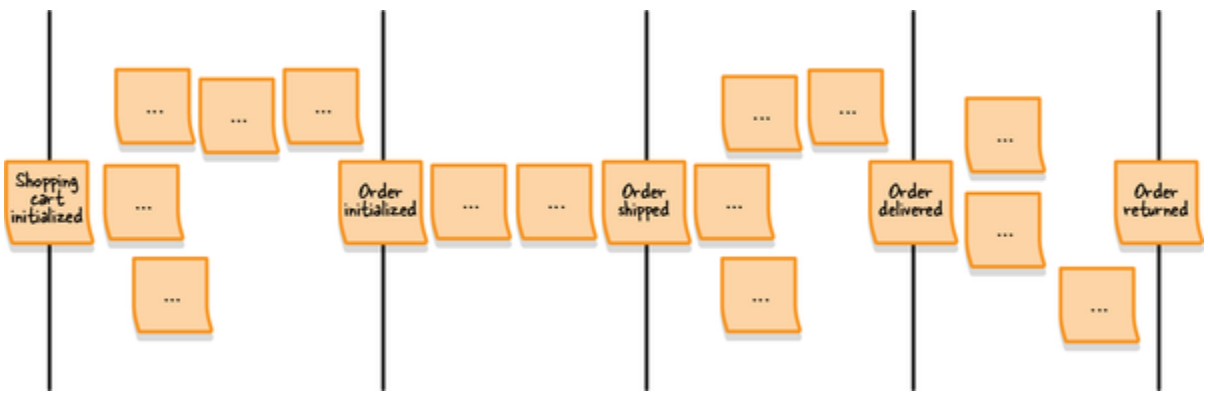
STEP 3: PAIN POINTS (CONTINUED)



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 4: PIVOTAL EVENTS

- Once you have a timeline of events augmented with pain points, look for significant business events indicating a change in context or phase.
- These are called pivotal events and are marked with a vertical bar dividing the events before and after the pivotal event.
- For example, “shopping cart initialized,” “order initialized,” “order shipped,” “order delivered,” and “order returned” represent significant changes in the process of making an order
- **Pivotal events are an indicator of potential bounded context boundaries**



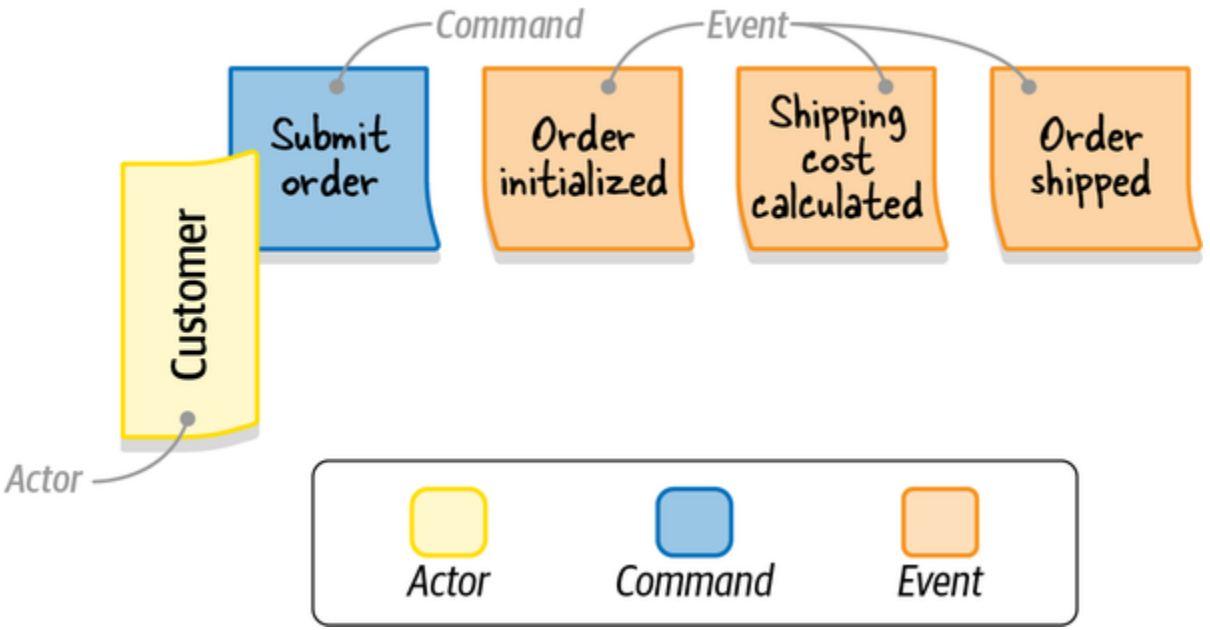
STEP 5: COMMANDS

- Whereas a domain event describes something that has already happened, a command describes what triggered the event or flow of events.
- Commands describe the system's operations and, contrary to domain events, are formulated in the imperative.
  - Publish campaign
  - Roll back transaction
  - Submit order

STEP 5: COMMANDS (CONTINUED)

- Commands are written on light blue sticky notes and placed on the modeling space before the events they can produce.
- If a particular command is executed by an actor in a specific role, the actor information is added to the command on a small yellow sticky note
- The actor represents a user persona within the business domain, such as customer, administrator, or editor.
- Naturally, not all commands will have an associated actor.
- Therefore, add the actor information only where it’s obvious

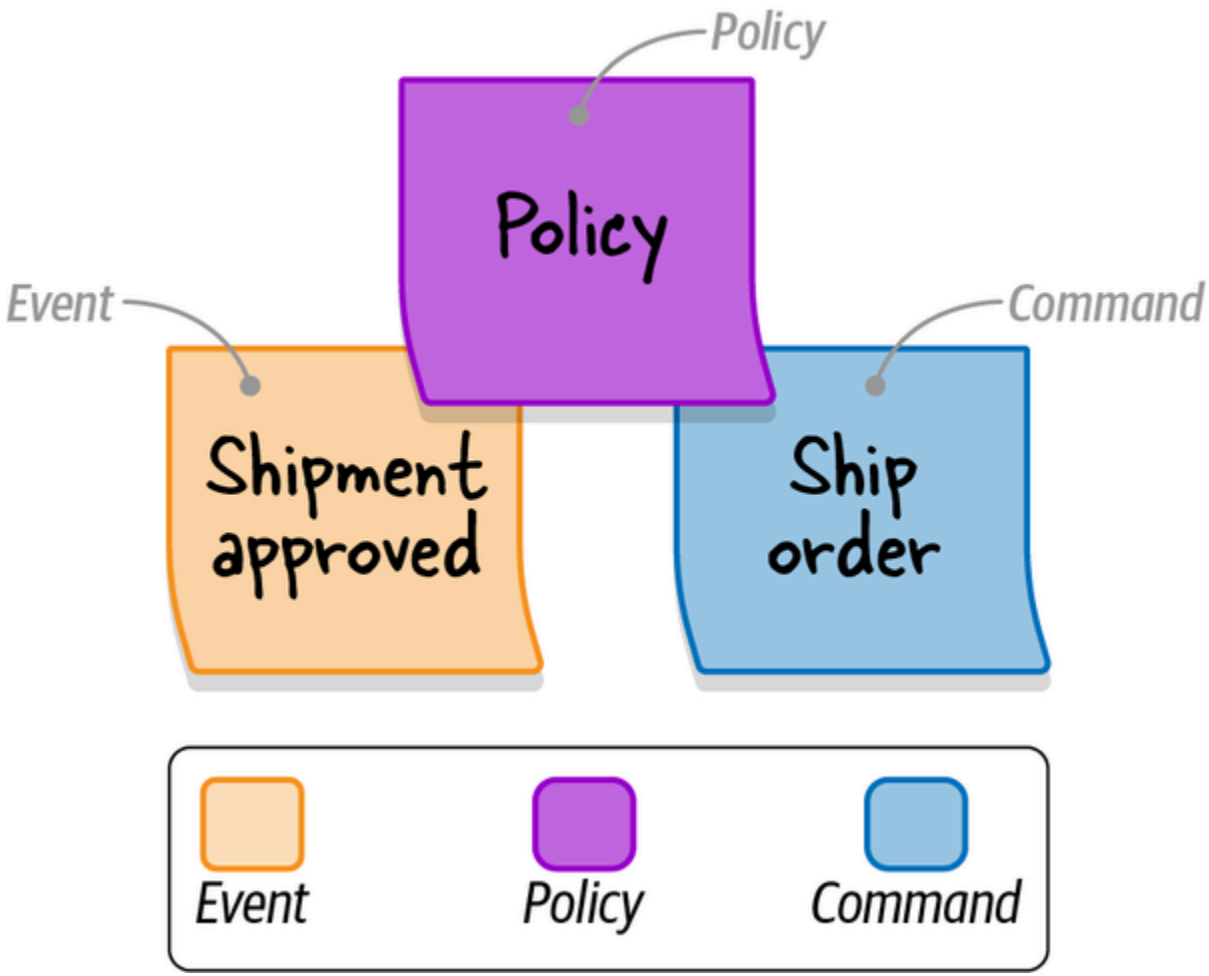
STEP 5: COMMANDS (CONTINUED)



STEP 6: POLICIES

- Some commands are added to the model but have no specific actor associated with them.
- During this step, you look for automation policies that might execute those commands.
- An *automation policy* is a scenario in which an event triggers the execution of a command
- Policies are represented as purple sticky notes connecting events to commands
- Example: You need to trigger the escalate command after the “complaint received” event, but only if the complaint was received from a VIP customer, you can explicitly state the “only for VIP customers” condition on the policy sticky

STEP 6: POLICIES (CONTINUED)

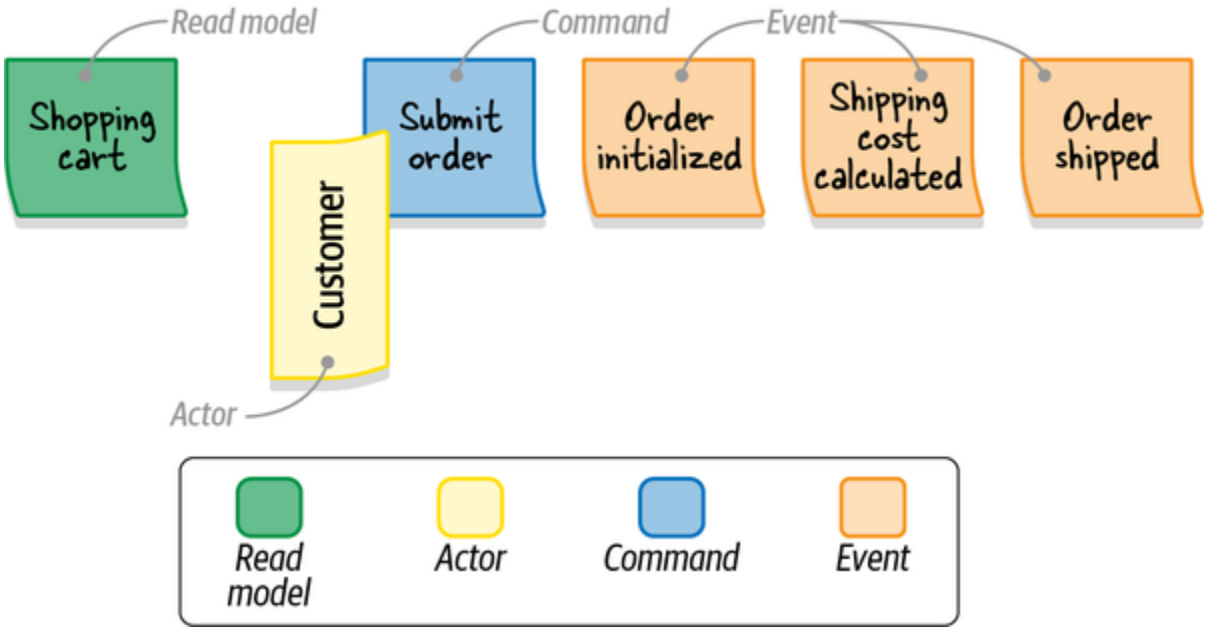


If the events and commands are far apart, you can draw an arrow on the modeling surface to connect them.

STEP 7: READ MODELS

- A read model is the view of data within the domain that the actor uses to make a decision to execute a command.
- This can be one of the system's screens, a report, a notification, and so on.
- The read models are represented by green sticky notes with a short description of the source of information needed to support the actor's decision
- Since a command is executed after the actor has viewed the read model, on the modeling surface the read models are positioned before the commands

STEP 7: READ MODELS (CONTINUED)

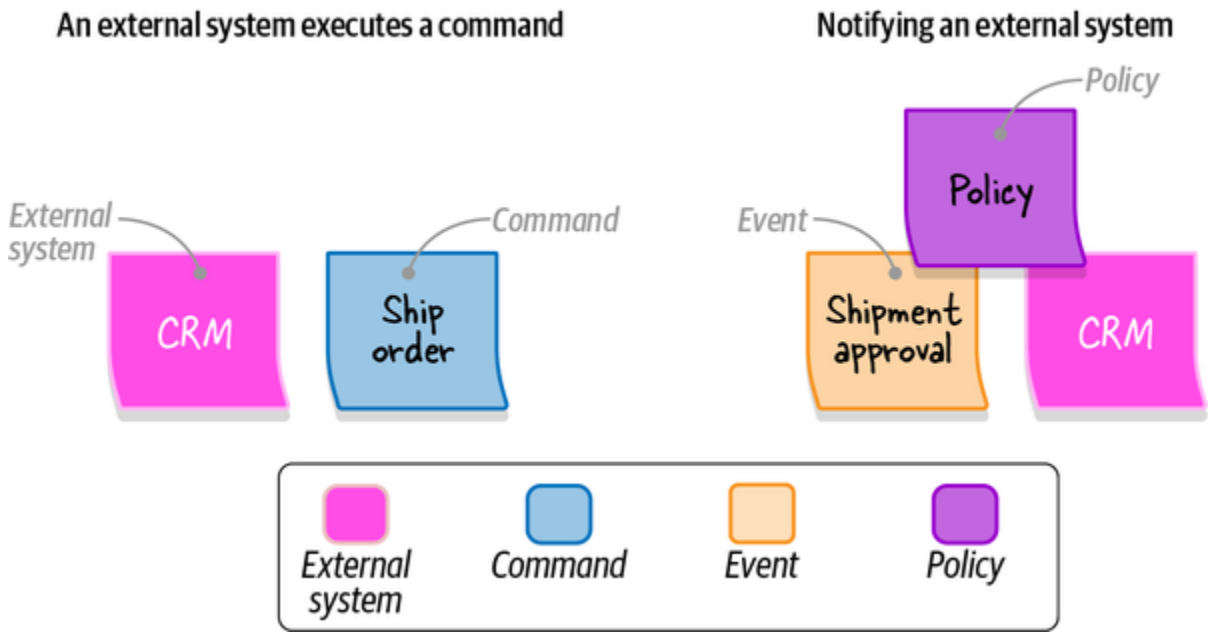


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 8: EXTERNAL SYSTEMS

- This step is about augmenting the model with external systems.
- An external system is defined as any system that is not a part of the domain being explored.
- It can execute commands (input) or can be notified about events (output).
- The external systems are represented by pink sticky notes.
- **By the end of this step, all commands should either be executed by actors, triggered by policies, or called by external systems**

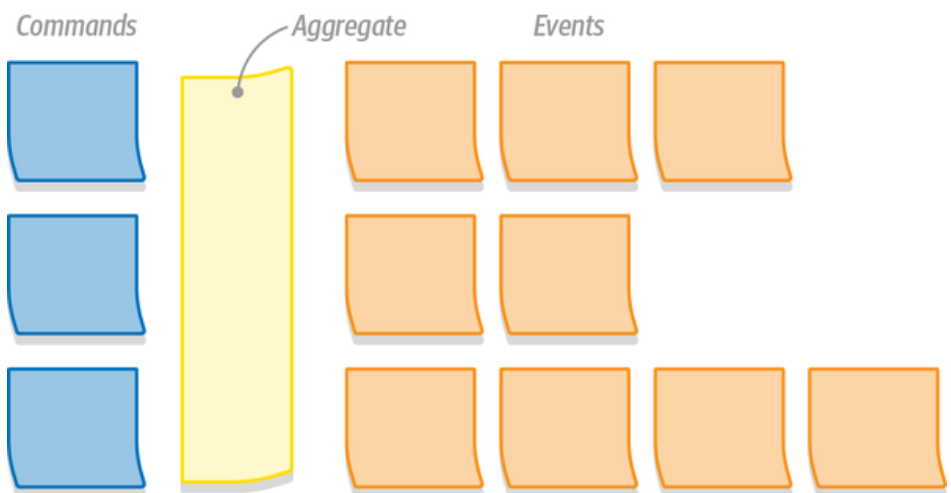
STEP 8: EXTERNAL SYSTEMS (CONTINUED)



Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 9: AGGREGATES

- Once all the events and commands are represented, the participants can start thinking about organizing related concepts in aggregates.
- An aggregate receives commands and produces events.
- Aggregates are represented as large yellow sticky notes, with commands on the left and events on the right

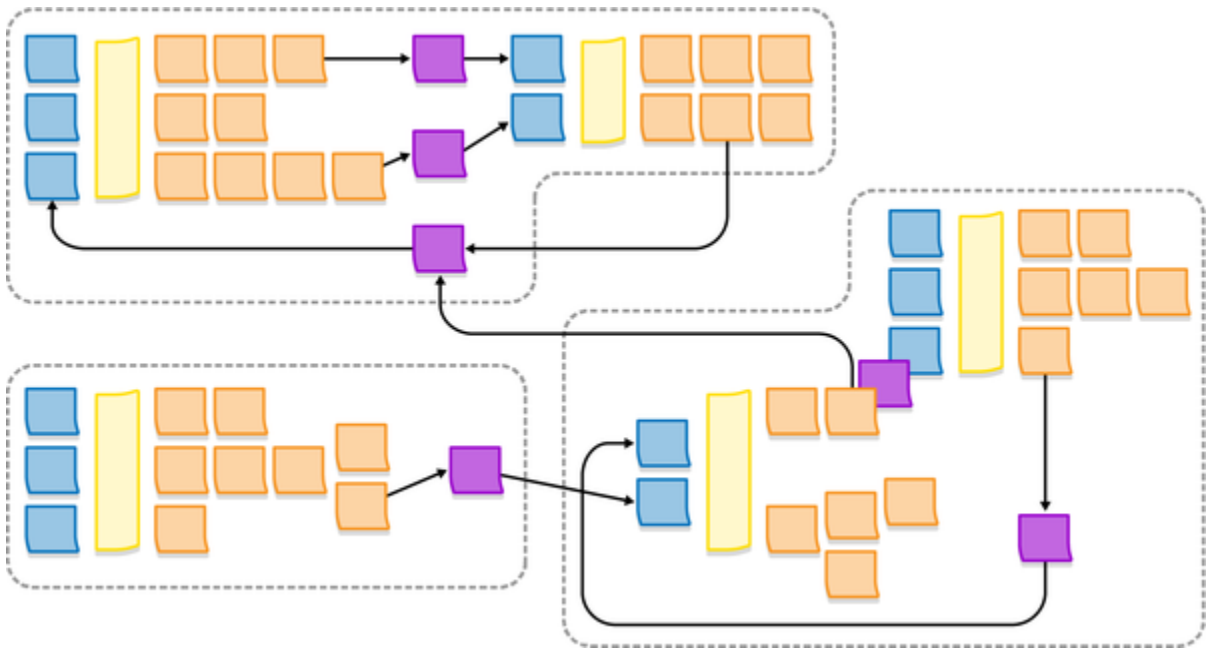


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

STEP 10: BOUNDED CONTEXTS

- The last step of an EventStorming session is to look for aggregates that are related to each other, either because they represent closely related functionality or because they’re coupled through policies
- The groups of aggregates form natural candidates for bounded contexts’ boundaries

STEP 10: BOUNDED CONTEXTS (CONTINUED)

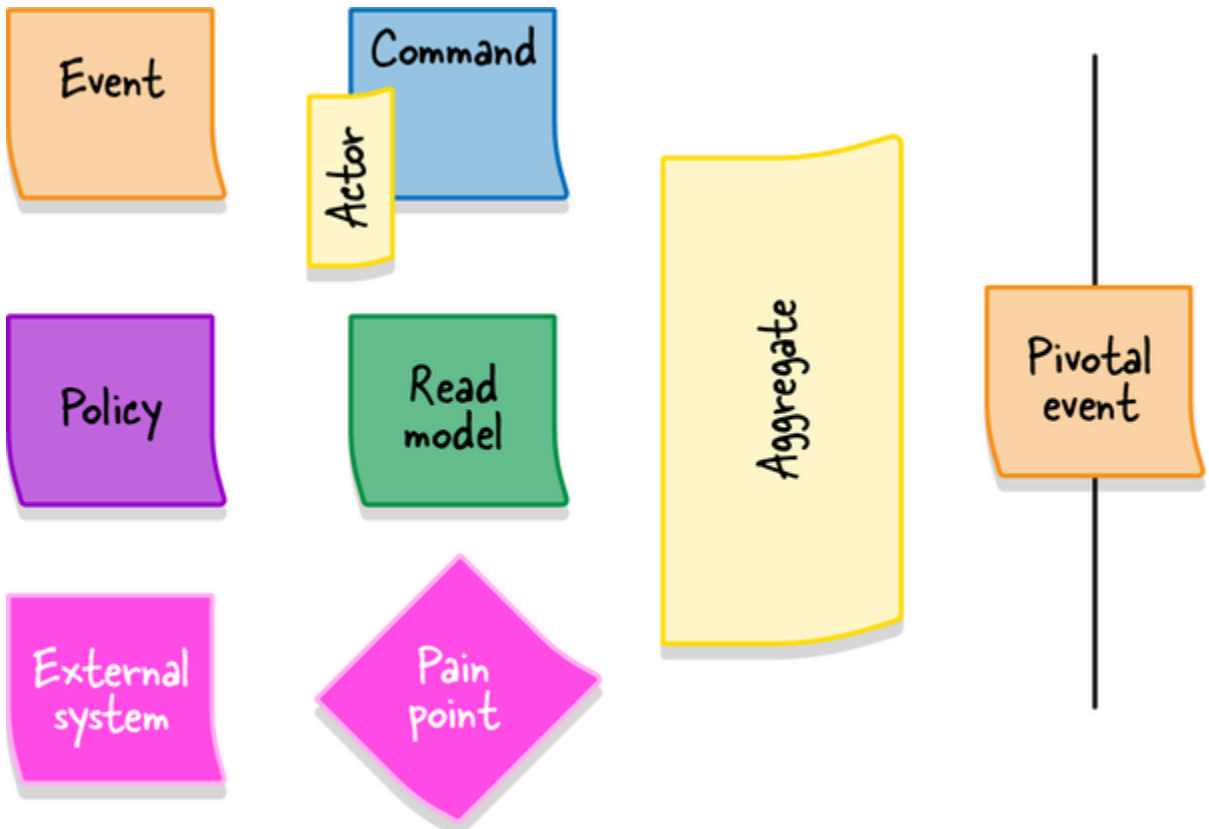


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

VARIATIONS

- Alberto Brandolini, the creator of the EventStorming workshop, defines the EventStorming process as guidance, not hard rules.
- You are free to experiment with the process to find the “recipe” that works best for you.
- The real value of an EventStorming session is the process itself—the sharing of knowledge among different stakeholders, alignment of their mental models of the business, discovery of conflicting models, and, last but not least, formulation of the ubiquitous language.

KEEP A LEGEND AROUND

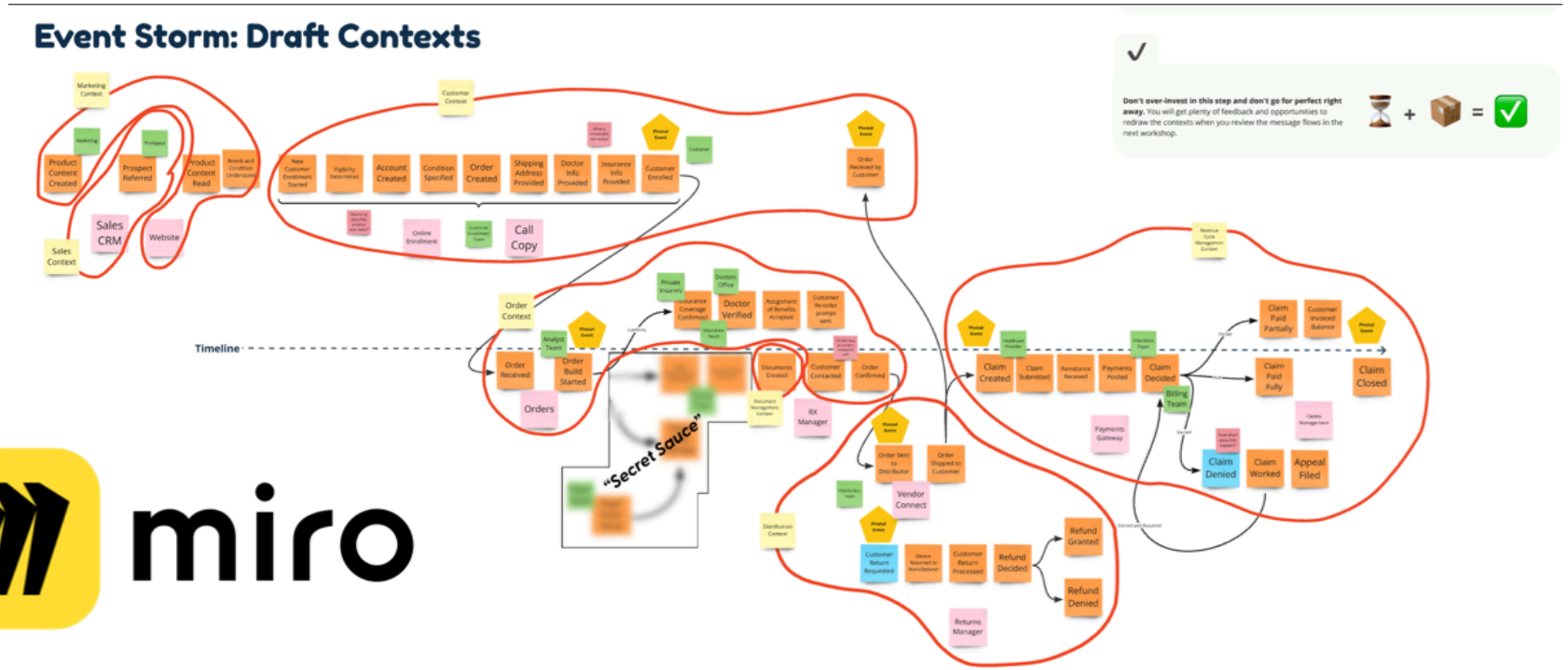


Learning Domain Driven Design - Vlad Khononov, Published by O'Reilly Media, Inc.

WATCH THE DYNAMICS

- Watch the energy. If there is slowdown reignite with questions or move to the next stage
- Ensure that the *entire group* is present. Involve even those that are reluctant to participate
- Ensure that you take breaks and ready for collaboration
- Five (Preferable) to Ten participants is optimal, if more needed break up into teams and compare notes

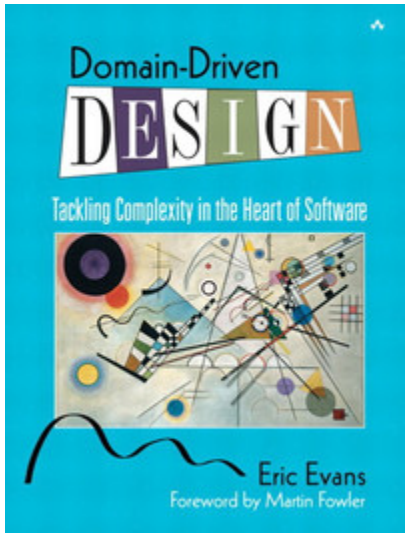
NEED IT ONLINE? MIRO



miro

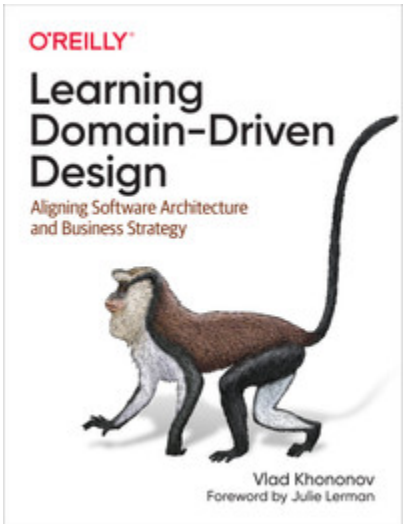

RESOURCES

DOMAIN DRIVEN DESIGN



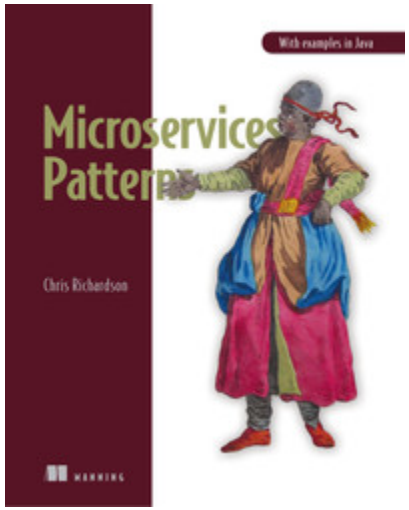
Eric Evans Published by Addison-Wesley Professional

LEARNING DOMAIN DRIVEN DESIGN



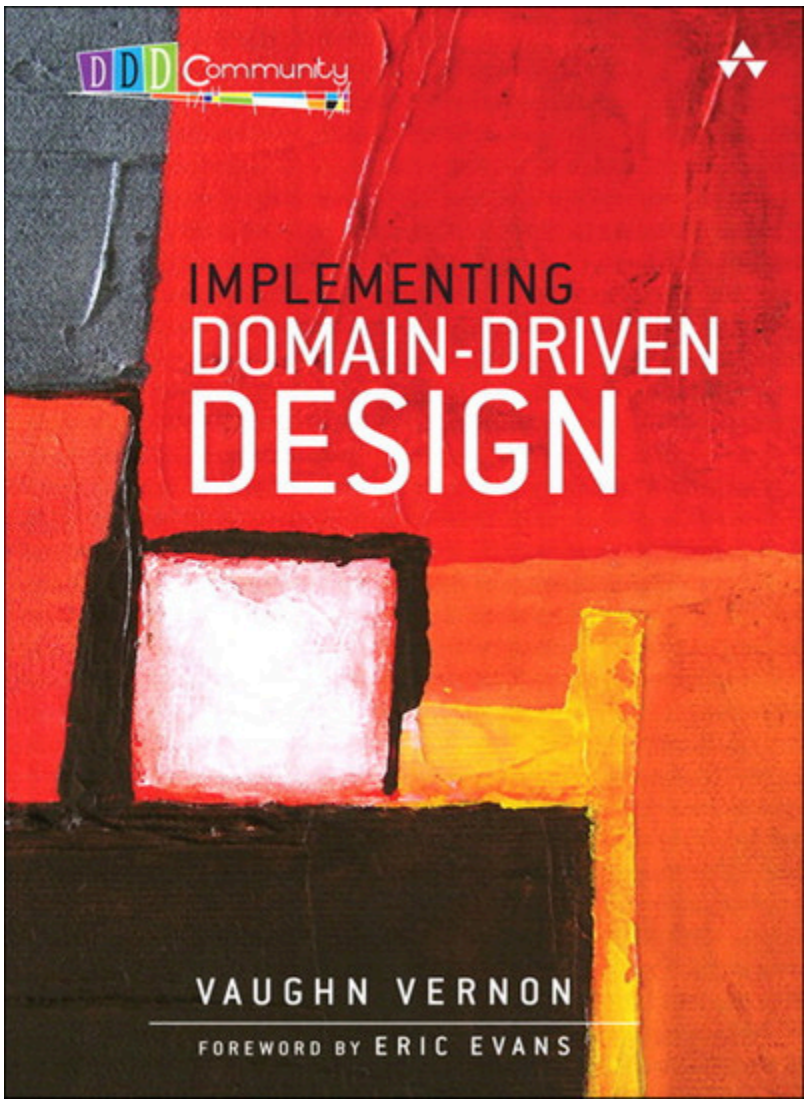
Vlad Khononov Published by O’Reilly Media, Inc.

MICROSERVICES PATTERNS



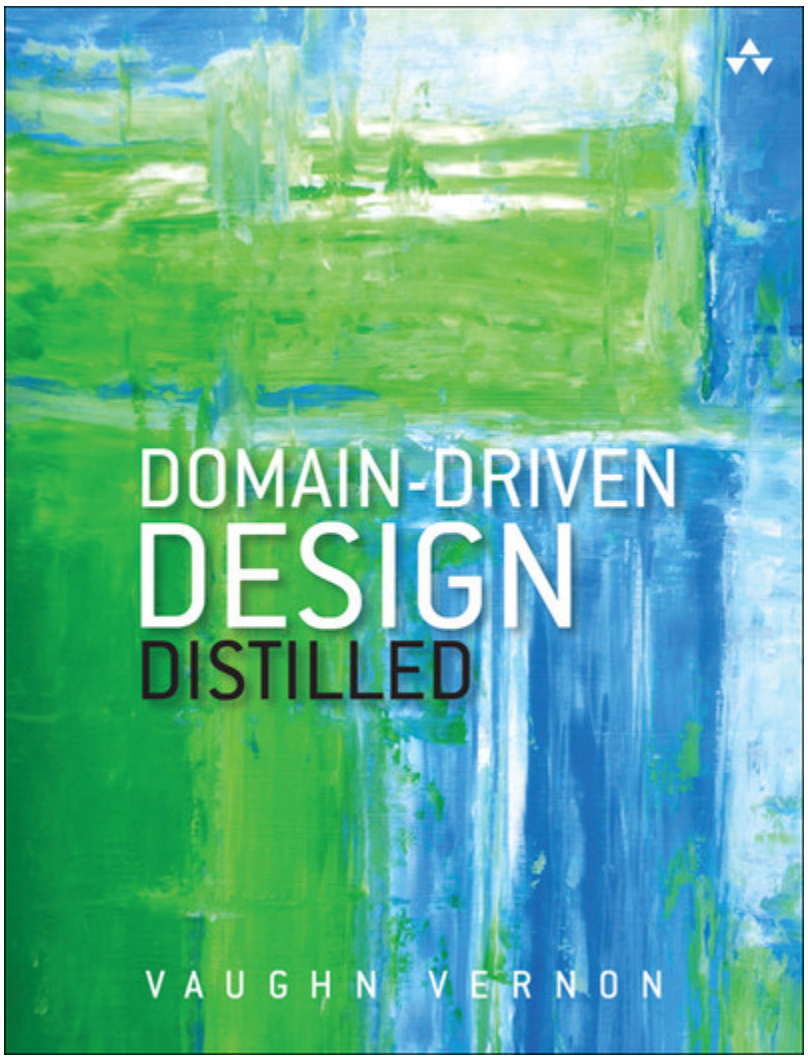
Chris Richardson Published by Manning Publications

IMPLEMENTING DOMAIN-DRIVEN DESIGN



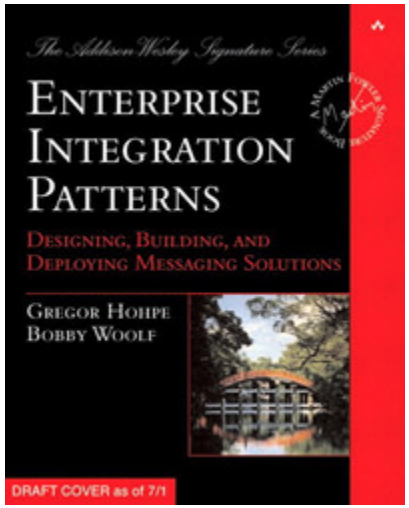
Vaughn Vernon Published by Addison-Wesley Professional

DOMAIN DRIVEN DESIGN DISTILLED



Vaughn Vernon Published by Addison-Wesley Professional

ENTERPRISE INTEGRATION PATTERNS



Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions Gregor Hohpe, Bobby Woolf Published by Addison-Wesley Professional

HOW AGILE CAN CRIPPLE EFFECTIVE DESIGN WORK (AND WHAT TO DO ABOUT IT)

Paul Rayner — How Agile Can Cripple Effective Design Work (and what to do about it)

MICROSERVICES.IO

Visit the https://microservices.io/ website